FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA

2003/2004

Voice Recognition

Preprocessing Voice Signal

By
YEOH LING TZE
WEK 010314

Supervisor
Mr. Yamani Idna b. Idris

Moderator
Mr. Amirrudin Hj Kamsin

Submission Date
19 February 2004

Abstract

Voice recognition is the process by which a computer is able to map an acoustic signal (i.e. spoken words) to the textual representation of those words. Voice recognition is also known as Speech Recognition, Speech Understanding System or Automatic Speech Recognition (ASR). Voice recognition techniques have been in the existence for approximately 30 years and many are currently achieving an accuracy rate of approximately 99 percent for isolated word recognition. Voice recognition work now run under the gamut from practical 24 hour a day applications of isolated word recognizers in industrial or governmental operations, to research and development work in versatile recognizers of complex spoken sentences. The number of commercial devices being sold is apparently expanding exponentially, public awareness is rising, and the prospects for future impact on human interaction with machines are very bright. Voice recognition is a part of a broader speech processing technology that also involve in computer identification and verification of speakers, computer synthesis of speech and production of stored spoken responses, computer analysis of the physical and psychological state of speaker, efficient transmission of spoken conversations, detection of speech pathologies and aids to the handicapped. Isolated word recognition has many applications in the areas of voice commanded machine operation, voice dialing, command and control systems, security control and hands free cellular telephone dialing.

Acknowledgement

I would like to thank the many friends who have helped me in my frequent struggles with various aspects of the subject of voice recognition. First and foremost, warm and grateful thanks go to Mr. Yamani Idna b. Idris as my supervisor for my final year project Voice Recognition. He has overcome my initial hesitations with his enthusiasm and he has nurtured my understanding with his amazing generosity and patience. I am very appreciating of the advices and encouragement given by him during the development of this project. And also thank to my moderator Encik Amirrudin Hj Kamsin for giving me advices and ideas to enhance my project. He has also assisted my development in innumerable other ways. These two have provided support in times of crisis and criticism in times of confidence.

Many students have worked with me on aspect of voice communication. I would like to mention in particular Don Kong, Lee Kuan, and Vincent. Many of the ideas stem from fascination conversation with my project partner, Tee Boon Siong. He provides a great deal of stimulation for my work. I am heavily obliged to my family for donating that most precious of all gifts – time – to help improve my viva presentation. I have been very fortunate in obtaining support of my voice recognition research from the main library of Malaya University and also from the library of Engineering Faculty of Malaya University.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 Project Introduction

Voice recognition technology has been in research and development for more than three decades. Voice recognition is a process transcribing speech into text automatically. Now this technology is close to becoming more practical. In other words, rather than keyboard interface, all computers are equipped with a voice interface. In such a world, the product inspector can enter data verbally, leaving his or her hands free to perform production operations. For this reason, a large amount of research and development is presently taking place in the field of voice recognition and understanding. Voice interfaces will gradually replace traditional keyboards. It will evolve with other high-technology fields, in particular the field of artificial intelligence.

1.2 Objectives

Below are the objectives of developing this project :-

a) Pre processing the voice signal

b) Develop a simple and yet completes voice recognition algorithm using zero crossing.

c) Replace the remote control with a microphone.

d) Help the disabled or the people who suffer attacks from diseases such as arthritis

e) To integrate the voice recognition algorithm into a FPGA board

1.3 Scope

The scope of the system will determine the range and how the system will work. The main scope of the project is to design a voice recognition system in terms of pre processing the voice signal and then implementing a best voice recognition algorithm and at the same time generating VHDL source code. Individual modules will be defined in terms of its functions and interconnections between them. The first phase of the project is about learning the fundamentals of voice recognition. During this phase, VHDL is also learned.

1.4 Chapter to Chapter Outline

In developing this system, a lot of stages have been gone through starting from drafting planning until the system is fully developed. This report will include every step that is taken into considerations while developing a voice recognition system that is divided to each different chapter outline.

Chapter 1 will bring us to the introduction of the world of voice recognition that is also known as speech recognition. The objectives and scopes of developing this system will be provided here. Chapter to chapter outline and project schedule are also provided in this chapter.

Chapter 2 will introduce us to the literature review of the system that will include the reviews of the existing design and techniques used to develop a voice recognition system.

This chapter will also touch a little bit about VHDL as a hardware description language and FPGA.

Chapter 3 cover the methodology used in developing the system. Project development techniques will be mentioned here, as well as the development tools used.

Chapter 4 is where the details of proposed design will be mentioned. The process of pre processing the voice signal starting from the input until the expected output will be explained in full details.

Chapter 2 : Literature Review

2.1 Definition of Voice Recognition

Voice Recognition is defined as the ability of a machine to recognize and understand spoken words. In details, voice recognition is the process of converting an acoustic signal, captured by the microphone to a set of words. The recognized words can be the final result, as for application such as command and controls, data entry and document writing.

The following definitions are the basics needed for understanding voice recognition technology.

- **Utterance**

An utterance is the vocalization (speaking) of a word or words that represent a single meaning to the computer. Utterances can be a single word, a few words, a sentence, or even multiple sentences.

- **Accuracy**

The ability of a recognizer can be examined by measuring its accuracy - or how well it recognizes utterances. This includes not only correctly identifying an utterance but also identifying if the spoken utterance is not in its vocabulary. Good systems have an accuracy of 98% or more. The acceptable accuracy of a system really depends on the application.

- **Speaker Dependence/Independence**

The first criterion is to consider *who* is going to use the system. The reason for this is that a voice recognition system is either speaker-dependent or speaker-independent. A

speaker-dependent system must be trained by the user. A training session will be required as to speak each word to be recognized into the system. In most cases, each word must be repeated at least five ties for the training session to be successful. The system must be retrained by different users and it is obviously time consuming. However, for many application a speaker-dependent system is the most practical and economical choice, since it is very accurate in recognizing a given user's voice. Meanwhile, a speaker-independent system does not require training and will respond to the voice input of any user. However, the trade-off is usually smaller vocabulary and increased software complexity. [Andrew, 1987]

- **Vocabulary**

Only a few vocabularies will be used in the recognizing process. The example: *zero, one,* and *two*. Many industrial and consumer application exist that could utilize this simple vocabulary. With today's technology, good voice recognition is directly related to small vocabularies. As the system vocabulary increases, the system memory complexity, response time and cost, all increase in direct proportion. Generally, smaller vocabularies are easier for a computer to recognize, while larger vocabularies are more difficult. [Andrew, 1987]

2.2 Types of Voice Recognition

Voice recognition systems can be separated in several different classes by describing what types of utterances they have the ability to recognize. These classes are based on the fact that one of the difficulties is the ability to determine when a speaker starts and

finishes an utterance. Most packages can fit into more than one class, depending on which mode they're using.

- **Isolated Word versus Connected Speech**

Voice recognition fall into one of two functional categories: isolated-word recognition or connected-speech understanding systems. As the name implies, isolated-word recognition systems are designed to recognize or verify a single-word utterances. The technology for these systems is well developed and as a result, many commercial products employ isolated-word recognition.

A speech signal that contains whole phrases and sentences is called connected or continuous, speech. The speech signal is therefore broken down in to individual word signals. First, it is difficult to determine the precise boundaries of an individual word within a connected speech signal. Speaker must make a 200-300 ms pause between words. Second, the signal for a given word in connected phrase does not closely resemble the signal for the same word if spoken separately. Finally, the pronunciation of a given word is affected by the words and punctuation around it. [Andrew, 1987]All these problems make connected-speech recognition a difficult task indeed.

- **Voice Recognition versus Speech Understanding**

Voice recognition is usually associated with the signal-matching or template, techniques used in isolated-word recognition. Speech understanding on the other hand, is usually associated with the artificial intelligence techniques used in connected-speech understanding system. Such techniques attempt to interpret the meaning of speech signals

using knowledge about the speech, rather than simply matching patterns as in voice recognition systems.

- **Category of Voice Recognition**

Table 2.1 : Category of voice recognition parameters with its ranges

Parameters	Range
Speaking Mode	Isolated words to continuous speech
Speaking Style	Read speech to spontaneous speech
Enrollment	Speaker-dependent to Speaker-independent
Vocabulary	Small (less than 20 words) to Large (more than 20 words)
Language Model	Finite-state to Context-sensitive

2.3 Uses and Applications

Although any task that involves interfacing with a computer can potentially use this system, the following applications are the most common right now.

Dictation

Dictation is the most common use for a voice recognition system today. This includes medical transcriptions, legal and business dictation, as well as general word processing. In some cases special vocabularies are used to increase the accuracy of the system.

Command and Control

Voice recognition systems that are designed to perform functions and actions on the system are defined as Command and Control systems. Utterances like "Open Netscape" and "Start" will do just that.

Telephony

Some systems allow callers to speak commands instead of pressing buttons to send specific tones.

Wearable

Because inputs are limited for wearable devices, speaking is a natural possibility.

Medical/Disabilities

Many people have difficulty typing due to physical limitations such as repetitive strain injuries (RSI), muscular dystrophy, and many others. For example, people with difficulty hearing could use a system connected to their telephone to convert the caller's voice to text.

Embedded Applications

Some newer cellular phones include voice recognition that allows utterances such as "Call Home".

2.4 Review of Existing Design

There are quite a number of voice recognition designs on the market that offer variety of functions and usages. Sensory's RSC-4x, Sensory's RSC-3x, Adelante's SRS-HDL

1.0R4 and ST Micro Electronics' Euterpe™ Digital Voice Processor are a few examples of voice recognition chip design that we can get from the market.

The RSC-4x is a voice/speech recognition and analog I/O mixed signal processor developed by Sensory Inc. Based on an 8-bit micro controller, the RSC- 4x integrates speech-optimized digital and analog processing blocks into a single chip solution capable of accurate speech recognition that may produce a high quality, low date-rate compressed speech and advanced music too.

2.4.1 Features

- **Full Range of Sensory Speech™ 7 Capabilities**

a) Enhanced word spotting capability (10 speaker independent or 5 speaker dependent words) in parallel

b) Noise robust speaker independent, dependent & continuous listening recognition

c) High quality, 3.7-7.8 kbps speech synthesis & sound effects

d) Speaker verification (SV) – voice biometric security

e) 8 voice MIDI-compatible music synthesis coincident with speech; drum track feature enables additional voices

f) Voice record & playback

g) Audio Wakeup from sleep

- **Integrated Single-Chip Solution**

a) 8-bit microcontroller

b) ROMless, 128KByte and 256KByte ROM options

c) 16 bit ADC, 10 bit DAC and microphone pre-amplifier

d) Independent, programmable Digital Filter engine

e) 4.8 KBytes total RAM (256Bytes "user" application RAM)

f) Five timers (3 GP, 1 Watchdog, 1 Multi Tasking)

g) Twin-DMA, Vector Math accelerator, and Multiplier

h) Built-in Analog Comparator Unit (4 inputs)

i) External memory bus: 20-bit Address (1Mbyte), 8-bit Data

j) On chip storage for SD, SV, templates (10 templates)

k) Code security through no ROM dump capability

l) Uses low cost 3.58MHz crystal (internal PLL)

m) Low EMI design for FCC and CE requirements

n) 24 configurable I/O lines with 10 mA (typical) outputs

o) Fully nested interrupt structure with up to 8 sources

p) Optional Real Time Clock

2.4.2 Block Diagram of RSC- 4x

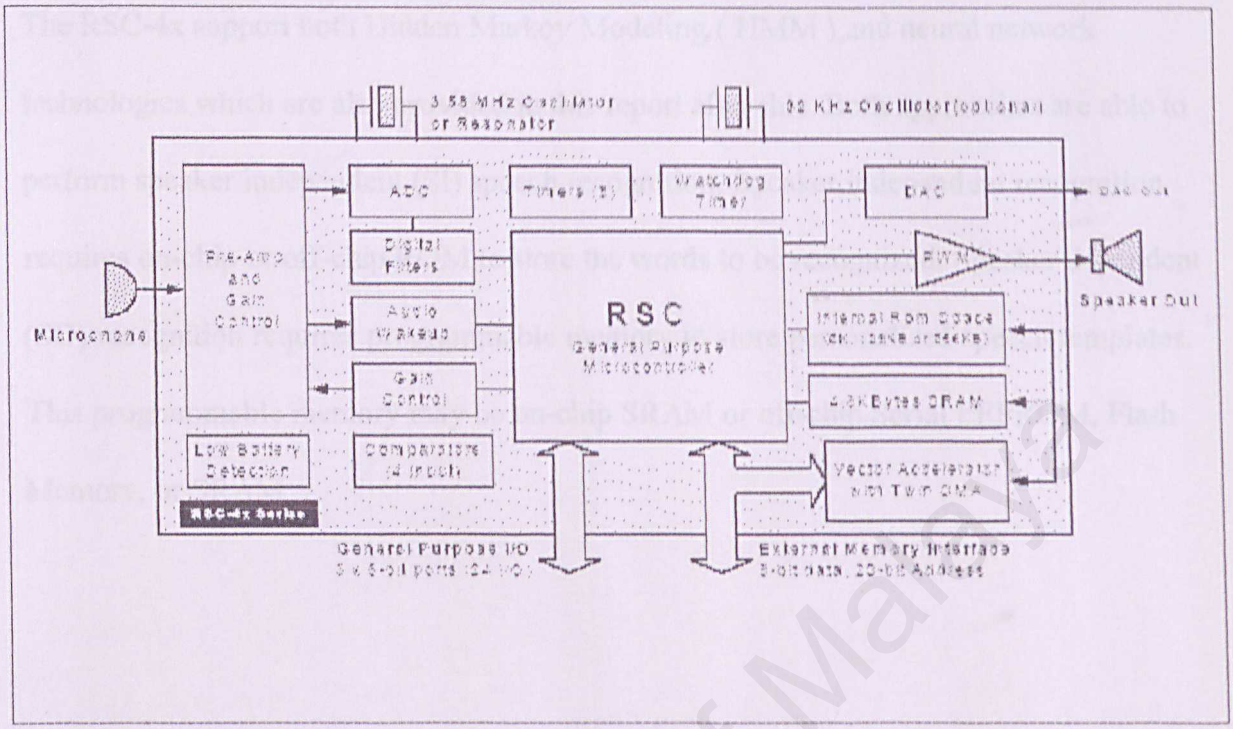


Figure 2.1 : The block diagram of the RSC-4x.

The RSC-4x features an eight-bit micro controller with on-chip ADC, DAC, preamplifier, RAM, ROM and optimized audio processing blocks. The CPU core embedded in the RSC-4x is an 8-bit, variable-length-instruction micro controller. The instruction set is similar to the 8051 micro controller, and has a variety of addressing mode, *MOV* and 16 bit instructions. The RSC- 4x processor avoids the limitations of dedicated A, B, and DPTR registers by having completely symmetrical sources and destinations for all instructions.

2.4.3 Algorithm used in RSC-4x

The RSC-4x support both Hidden Markov Modeling (HMM) and neural network technologies which are also provided in this report after this. Both approaches are able to perform speaker independent (SI) speech recognition. Speaker independent recognition requires on-chip or off-chip ROM to store the words to be recognized. Speaker dependent (SD) recognition requires programmable memory to store personalized speech templates. This programmable memory may be on-chip SRAM or off-chip Serial EEPROM, Flash Memory, or SRAM.

2.4.4 RSC-4x Architecture

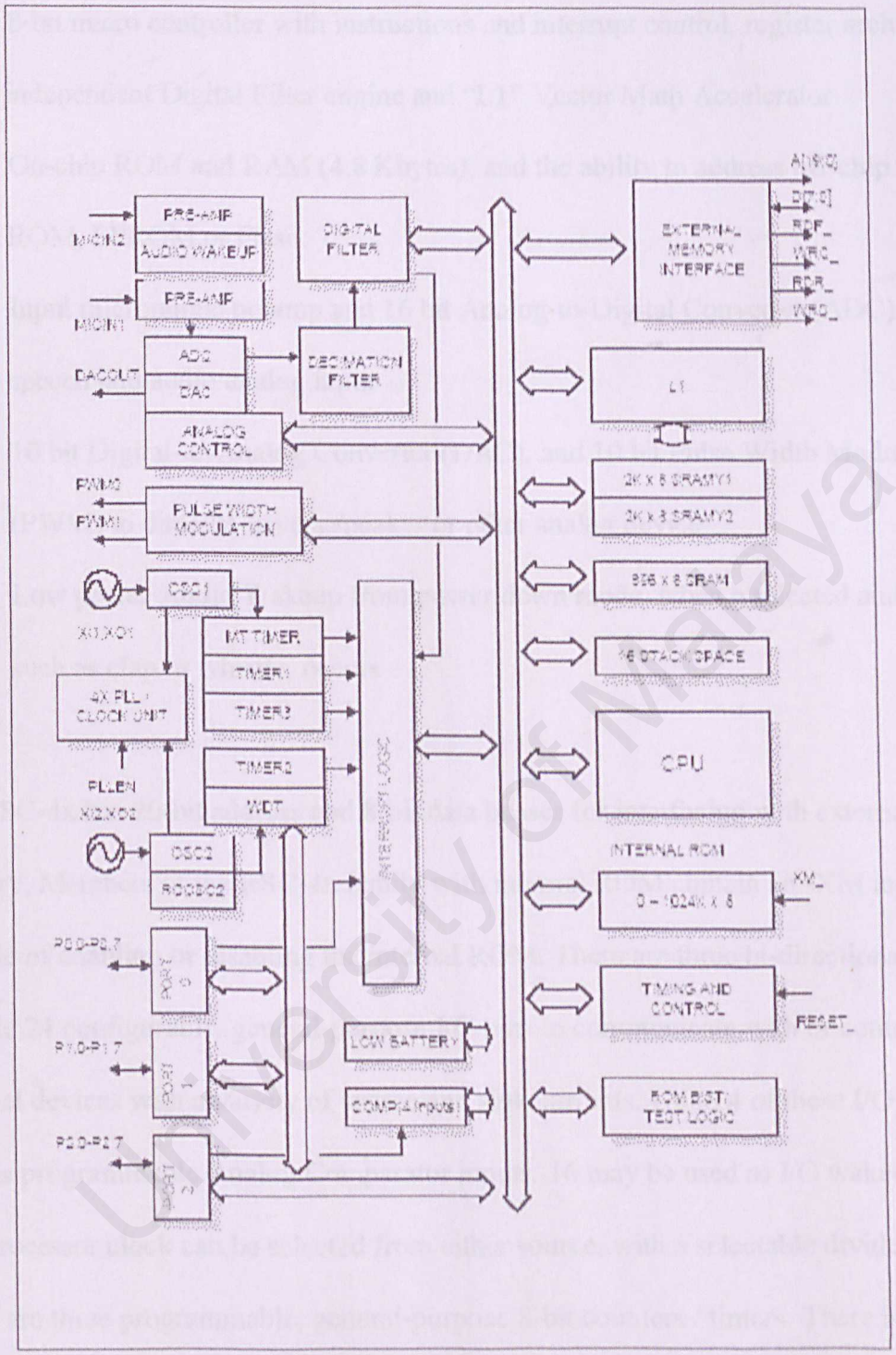


Figure 2.2 : RSC-4x Internal Block Diagram

The RSC-4x combines:

a) 8-bit micro controller with instructions and interrupt control, register architecture, independent Digital Filter engine and "L1" Vector Math Accelerator

b) On-chip ROM and RAM (4.8 Kbytes), and the ability to address off-chip RAM, ROM, EPROM or Flash.

c) Input microphone preamp and 16 bit Analog-to-Digital Converter (ADC) for speech and audio/analog input

d) 10 bit Digital-to-Analog Converter (DAC), and 10 bit Pulse Width Modulator (PWM) to directly drive a speaker or other analog device

e) Low power Audio Wakeup from power down mode, when a selected audio event, such as clap or whistle, occurs

The RSC-4x has 20-bit address and 8-bit data busses for interfacing with external memory. Members of the RSC-4x family with internal ROM contain an -XM input pin capable of enabling or disabling the internal ROM. There are three bi-directional ports provide 24 configurable, general-purpose I/O pins to communicate with or control external devices with a variety of source and sink currents. Up to 4 of these I/O may be used as programmable Analog Comparator inputs. 16 may be used as I/O wakeup. The processor clock can be selected from either source, with a selectable divider value. There are three programmable, general-purpose 8-bit counters / timers. There is also a Watchdog timer that may be used to exit an undesired condition in program flow, and multi-tasking timer to allow chip operations to share resources in parallel.

2.4.5 How RSC-4 Works in Recognizing Voice

An external microphone passes an audio signal to the preamplifier and ADC to convert the incoming speech signal into digital data. Speech features are extracted using the Digital Filter engine. The micro controller CPU processes these speech features using speech recognition algorithms in firmware, with the help of the "L1" Vector Accelerator and instruction set. The resulting speech recognition results may be used to control the consumer product application code, or to output speech or audio in the form of a dialog with the user of the consumer product. If desired, the output speech or audio signal from the RSC-4x is generated by a DAC for external amplification into a speaker, or a PWM capable of directly driving a speaker at typical consumer product volumes.

2.5 Review of Existing Technique Approach

There are many techniques have been implemented by the voice recognition system developer to extract the features and characteristic of the speech and convert it into understandable words. Each and every technique requires a lot of mathematical calculations that involve specific algorithm.

2.5.1 Template Matching

Template matching is one of the proven voice recognition techniques and has resulted in many low-cost commercial systems. This system must be trained by the system user. The associated software performs two functions during its training mode: acoustic analysis and template generation. In the recognition mode, the input signal is converted and acoustically analyzed as in the training

mode. The coded (linear predictive coding, LPC) data are then temporarily stored in matrix form for comparison to the reference templates that were generated during the training mode. A mathematical algorithm called *dynamic programming* can be used to compare templates.

Dynamic programming and a related technique called *time warping* are used to reduce recognition errors due to improper time alignment. [Andrew, 1987] Dynamic programming is a pattern-matching algorithm that is used for both voice recognition and visual pattern recognition. It is a matrix analysis technique that computes all the possible combinations of time alignments between the reference and unknown templates, the result being the best match between the two templates.

2.5.2 HMM-Based

In HMM*(Hidden Markov Model)* based voice recognition, estimation of parameters of HMMs is viewed as counterpart of training or learning in traditional sequential pattern recognition since speech signal can be represented by a sequence of *n*-dimension vectors after features are extracted from the speech signal. [Bahl, 1986] Voice samples with different duration contribute differently to estimation of parameters of the same HMM and as a result, HMM model some speech units well and others bad. Consequently, the confusion among speech units is unavoidable. The fact that on the whole, duration of phones varies regularly with context and speakers, indicates that enlarging training set alone is

not a perfect solution, although it is an available approach to the above problem because of somehow stochastic variation of duration of phones. Furthermore, while only smaller training set is accessible, for instance, in the case of speaker adaptation, the problem becomes very serious.

2.5.3 Word Trigram Models

Another widely used voice recognition technique is the algorithm using Word Trigram models. Word Trigram model is based on *Viterbi search* or known as one-pass DP. Among the many speech recognition algorithms, the Viterbi search is well suited for Hidden Markov Model (HMMs) used as language models.

2.5.4 Neural Network approach

The neural network is trained on a small amount of stereo speech data, composed of simultaneous close-talking and distant-talking samples. [Hirschberg, 1993] This small amount of adaptation data is negligible compared to the hours of data required to retrain the voice recognizer for a specific environment. Performance measurements on continuous voice recognition show that the system is capable of elevating the recognition accuracy of the voice recognizers to an acceptable operational level when they are used for distant-talking in noisy and reverberant environments

2.5.5 Acoustic Phonetic Analysis

The acoustic phonetic analysis is a logical strategy in order to accomplish the goal of voice recognition. This model is developed to handle large-vocabulary, speaker-independent, isolated-word recognition. With phonetic analysis, the speech sounds are divided into several broad phonetic categories. The template matching techniques are not as reliable as speaker-independent because of the signal variation from speaker to speaker.

The idea behind acoustic phonetic analysis is to analyze the speech or voice signal and separate the phoneme sounds to produce a detailed phoneme representation of the utterance. From the phonetic representation, a memory lookup operation would produce the corresponding word. Broad phonetic classifications such as vowel, nasal, and the fricative are less sensitive to fine phonetic variances from speaker to speaker. [Andrew, 1987] The idea is to interpret the spoken word into a series of broad phonetic categories. This information is then used to determine a small set of possible word candidates, from which a more detailed phonetic analysis produces the final word selection. Broad phonetic classification is a top-down reasoning strategy employed by fine phonetic analysis.

A closer look at a typical feature-analysis system with all the sounds of speech/voice that are divided into six broad phonetic categories :

Pure voiced vowel (V)	:	a, e, i, o, u, uh, aa, ee, er, uu, ar, aw
Nasal (N)	:	m, n, ng
Voiced fricative (VF)	:	z, zh, v, dh
Unvoiced fricative (UF)	:	s, sh, f, th
Plosive (P)	:	b, d, g, p, t, k, h
Glide (G)	:	r, w, l, y

Figure 2.3: Phonetic Categories for a typical Feature Analysis system

Now consider the words *yes, no, begin,* and *stop.* Using the six categories of **Figure 2.3**, the sounds of each word can be expressed as a sequence of sound categories as shown in **Figure 2.4**. But before that, the pronunciation of each word will be looked up from a dictionary.

Word	Dictionary Pronunciation	Phonetic Sound Sequence
Yes	yes	G - V – UF
No	nō	N – V
Begin	bigin	P – V – P – V - N
Stop	stop	UF - P – V – P

Figure 2.4: Sound Category Sequence for *yes, no, begin* and *stop.*

Assume that the vocabulary of our speaker-independent system consists solely of the four words. A speaker utters one of the words into the system, and an analysis algorithm translates the speech signal into one of the phonetic sequences in **Figure 2.4**. Next, a

decision-making algorithm must be used to determine which of the four words was actually spoken.

2.6 VHDL

VHDL is an acronym which stands for VHSIC Hardware Description Language. VHSIC is yet another acronym which stands for Very High Speed Integrated Circuits. VHDL describes hardware much the same way as schematics. VHDL is just another way of describing what outputs of a digital circuit are desired when it is given certain inputs. The critical difference between VHDL and these other languages are that it can be readily interpreted by software, enabling the computer to accomplish our design. . It is used to describe digital hardware in an abstract (and therefore easily changeable) way.

It is being used for documentation, verification, and synthesis of large digital designs. The same VHDL code can theoretically achieve all three of these goals, thus saving a lot of effort. VHDL is designed to fill a number of needs in the design process. Firstly, it allows description of the structure of a design, that is how it is decomposed into sub-designs, and how those sub-designs are interconnected. Secondly, it allows the specification of the function of designs using familiar programming language forms. Thirdly, as a result, it allows a design to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping. [Peter, 1990]

VHDL can be used to take three different approaches to describing hardware. These three different approaches are the structural, data flow, and behavioral methods of hardware description. Most of the time a mixture of the three methods is employed.

2.6.1 Primary Design Unit Model Structures

Each VHDL design unit comprises an "entity" declaration and one or more "architectures". Each architecture defines a different implementation or model of a given design unit. The entity definition defines the inputs to, and outputs from the module, and any "generic" parameters used by the different implementations of the module.

Entity Interface Specifications

```
entity name is
    port( port definition list );-- input/output signal ports
    generic( generic list);  -- optional generic list
  end name;
```

Explanation:

1) An **in** port can be read but not updated within the module, carrying information into the module.
2) An **out** port can be updated but not read within the module, carrying information out of the module.
3) A **buffer** port likewise carries information out of a module, but can be both updated and read within the module.
4) An **inout** port is bidirectional and can be both read and updated, with multiple update sources possible.

5) **Generics** allow static information to be communicated to a block from its environment for all architectures of a design unit. These include timing information (setup, hold, delay times), part sizes, and other parameters.

- **Architecture**

Once an entity has had its interface specified in an entity declaration, one or more implementations of the entity can be described in architecture bodies. Each architecture body can describe a different view of the entity. Architecture defines one particular implementation of a design unit, at some desired level of abstraction.

```
architecture arch_name of entity_name is
    ... declarations ...
begin
    ... concurrent statements ...
end
```

Declarations include data types, constants, signals, files, components, attributes, subprograms, and other information to be used in the implementation description. [Douglas, 1998]

Concurrent statements describe a design unit at one or more levels of modeling abstraction, including dataflow, structure, and/or behavior.

- **Behavioral Model:** No structure or technology implied. Usually written in sequential, procedural style.
- **Dataflow Model:** All data path shown, plus all control signals.
- **Structural Model:** Interconnection of components.

- **Signals**

Signals are used to connect sub modules in a design. Signals are declared via signal declaration statements or entity port definitions, and may be of any data type. The declaration syntax is:

```
signal sig_name: data_type [:=initial_value];
```

Ports of an object are treated exactly as signals within that object.

2.6.2 VHDL PACKAGES

A VHDL *package* contains subprograms, constant definitions, and/or type definitions to be used throughout one or more design units. Each package comprises a "declaration section", in which the available (i.e. exportable) subprograms, constants, and types are declared, and a "package body", in which the subprogram implementations are defined, along with any internally-used constants and types. The declaration section represents the portion of the package that is "visible" to the user of that package. The actual implementations of subroutines in the package are typically not of interest to the users of those subroutines.

- **Package component and subprogram declarations**

```
package package_name is
   ... exported constant declarations
   ... exported type declarations
   ... exported subprogram declarations
  end package_name;

package body package_name is
  ...
  ... type definitions
  ... subprograms
end package_name
```

  - **Package declaration**, which defines its interface,
  - **Package body**, which defines the deferred details. The body part may be omitted if there are no deferred details.

- **VHDL Standard Packages**

  a) STANDARD - basic type declarations (always visible by default)

  b) TEXTIO - ASCII input/output data types and subprograms. [Morris, 2000]

- **IEEE Standard 1164 Package**

This package contained in the 'ieee' library supports multi-valued logic signals with type declarations and functions.

```
library ieee;      -- VHDL Library stmt
```

2.6.3 Design Units and Libraries

A number of VHDL constructs may be separately analyzed for inclusion in a design library. These constructs are called *library units*. The *primary* library units are entity declarations, package declarations and configuration declarations. A design file may contain a number of library units. The structure of a design file can be specified by the syntax:

```
design_file ::= design_unit { design_unit }
design_unit ::= context_clause library_unit
context_clause ::= { context_item }
context_item ::= library_clause | use_clause
library_clause ::= library logical_name_list ;
logical_name_list ::= logical_name { , logical_name }
library_unit ::= primary_unit | secondary_unit
primary_unit ::=
entity_declaration | configuration_declaration | package_declaration
secondary_unit ::= architecture_body | package_body
```

Libraries are referred to using identifiers called logical names. This name must be translated by the host operating system into an implementation dependent storage name. Library units in a given library can be referred to by prefixing their name with the library logical name. So for example, ttl_lib.ttl_10 would refer to the unit ttl_10 in library ttl_lib.

2.6.4 Advantages of using VHDL

There are many reasons why it makes good design sense to use VHDL:

- **Portability**: Technology changes so quickly in the digital industry that discrete digital devices require constant rework in order to remain current. VHDL is designed to be device-independent, meaning that if we describe our circuit in VHDL, as opposed to designing it with discrete devices, changing hardware becomes a (relatively) trivial process

- **Flexibility**: Changes of design specification can't be helped. Design work is usually focused on creating small, easily maintainable components and then integrating these components into a larger device. On larger projects different teams of engineers will each design separate parts of the project at the same time. This can mean that if one component in the project changes, all of the components must change, even those being worked on by other engineering teams. But, if our design is using VHDL, all we have to do is change our code and we do not have to start over from the scratch.

- **Standard:** Many hardware description languages are developed to serve the simulators that run them. Some target a particular technology, design level or

methodology. VHDL is independent of technology, thus reduces confusion in making the interfaces between tools, companies and product easier.

- **Cost:** VHDL make the most reliable design process, with minimum cost and time.
- **Productivity:** VHDL can increase productivity by shorten the time to market. Behavioral simulation can reduce design time by allowing design problems to be detected early on, avoiding the need to rework designs at gate level
- **Better design:** Behavioral simulation permits design optimization by exploring alternative architectures, resulting in better designs.
- **Reusability:** System may be used again in other instances for which it may or may not have been specifically intended. To move a design to a new technology, a specification needs not to start from scratch or reverse-engineer. Instead, the design tree to a behavioral VHDL description can be implemented in the new technology knowing that the correct functionality preserved.

2.7 FPGA

A field-programmable gate array (FPGA) is an integrated circuit (IC) that can be programmed in the field after manufacture. FPGAs are used by engineers in the design of specialized ICs that can later be produced hard-wired in large quantities for distribution to computer manufacturers and end users.

FPGA contain hundreds (or thousands) of Configurable Logic Block (CLB). One CLB is a rectangular area on the chip that contains a lookup table (LUT), a flip-flop and routing. LUT are used to create a logic function such as AND, OR, XOR. See **Figure 2.5**. The

flip-flop allows synchronization (based on a clock signal) and the routing is just a lot of interconnection wiring between the CLB which can be linked together to form complex logic implementations.

Array of logic blocks is surrounded by programmable I/O blocks and connected with programmable interconnection .Most FPGAs do not provide 100% interconnection between their logic blocks because it would be prohibitively expensive. Instead, sophisticated software places and routes the logic on the device almost the same as a Printed Circuit Board (PCB) auto router would place and route components.

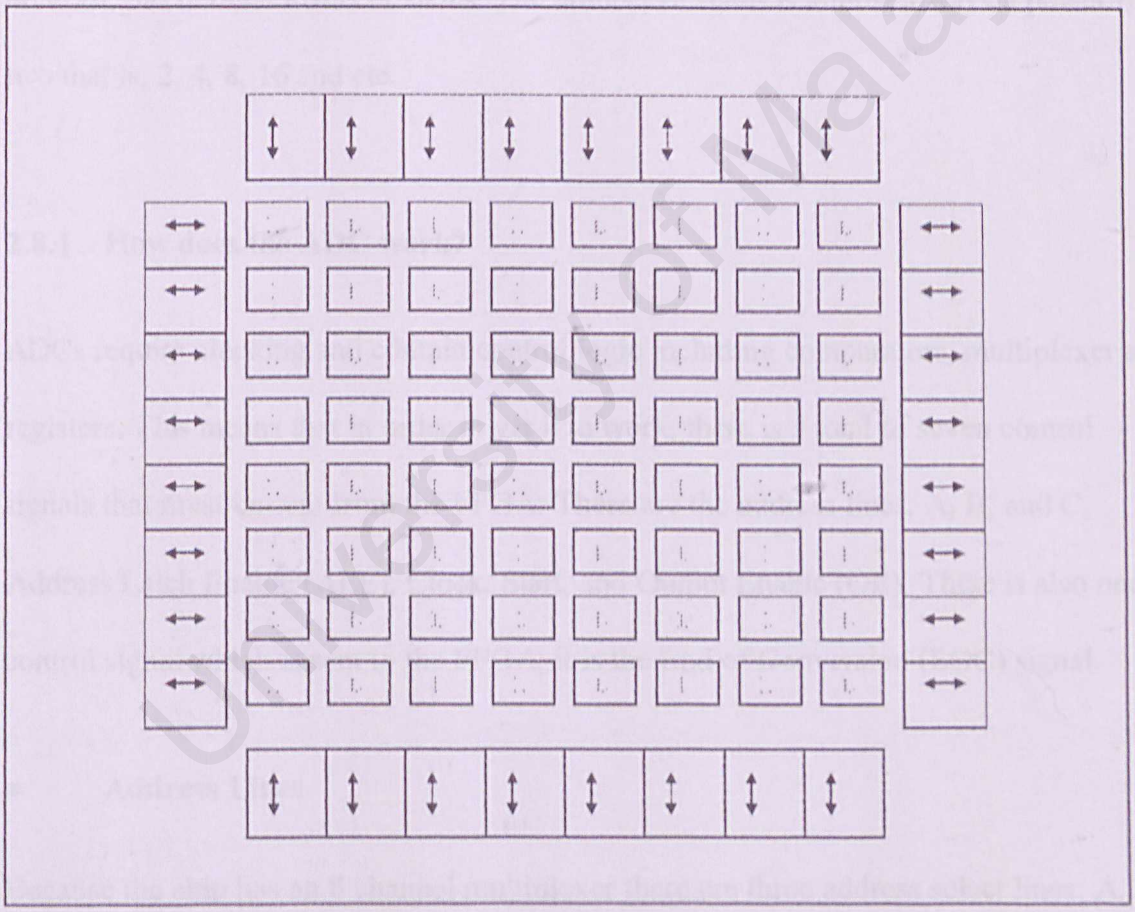


Figure 2.5 : Field Programmable Gate Arrays (FPGA)

2.8 Analog-to-Digital Converter

Analog-to-digital conversion is an electronic process in which a continuously variable (analog) signal is changed, without altering its essential content, into a multi-level (digital) signal.

ADC is used to convert an analog input voltage into a digital output. The input to an analog-to-digital converter (ADC) consists of a voltage that varies among a theoretically infinite number of values. Examples are sine waves, the waveforms representing human speech, and the signals from a conventional television camera. The output of the ADC, in contrast, has defined levels or states. The number of states is almost always a power of two that is, 2, 4, 8, 16 and etc.

2.8.1 How does the ADC work?

ADCs require clocking and contain control logic including comparators, multiplexer and registers. This means that in order to get it to work, there is a total of seven control signals that must be sent from the FPGA. These are the address lines, A, B, and C, Address Latch Enable (ALE), Clock, Start, and Output Enable (OE). There is also one control signal which is sent to the FPGA, it is the End of Conversion (EOC) signal.

- **Address Lines**

Because the chip has an 8 channel multiplexer there are three address select lines: A, B, and C. C is the most significant bit and A is the least . See **Table 2.2** for details.

Selected Analog Channel	Address Line C	B	A
IN0	L	L	L
IN1	L	L	H
IN2	L	H	L
IN3	L	H	H
IN4	H	L	L
IN5	H	L	H
IN6	H	H	L
IN7	H	H	H

Table 2.2: Channel Selection

- ALE

ALE is required to load the selected address lines into the ADC. Once loaded the multiplexer sends the appropriate channel to the converter on the chip. As with all control signals it is required to have an input value of Vcc - 1.5 up to 15V for a high and 1.5V down to -0.3V for a low. The following control signals are used to control the conversion.

- **Clock**

The clock signal is required to cycle through the comparator stages to do the conversion. There are 8, 8 clock cycle periods required in order to complete an entire conversion. This means that an entire conversion takes at least 64 clock cycles. (Up to 72 if the start

signal is received in the middle of an 8 clock cycle period.) The clock should conform to the same range as all other control signals. The maximum clock frequency is affected by the source impedance of the analog inputs.

- Start

The purpose of the start signal is to fold. On the rising edge of the pulse the internal registers are cleared and on the falling edge of the pulse the conversion is initiated. As clock speeds greater than that the user must make certain that enough time has passed since the ALE signal was pulsed so that the correct address is loaded into the multiplexer before a conversion begins.

- OE

The Output Enable signal causes the ADC to actually output the digital values on the output lines. The ADC stores the data in a tri-state output latch until the next conversion is started, but the data is only output when enabled.

- EOC

The End of Conversion signal is sent to the FPGA from the ADC. The signal goes low once a conversion is initiated by the start signal and remains low until a conversion is complete.

Chapter 3 : Project Methodology

3.1 Introduction

VHDL hierarchical design approach consists of both top-down and bottom-up design. Project can be implemented by either one methodology depending on the system requirements. Each approach has its own advantage that is useful to the system designer. In gathering information for developing the project, many techniques will be practiced such as discussions, surfing the World Wide Web and also brainstorming. The main development tools that have been used in developing a voice recognition system are XStend Board version 1.3.2 by Xess Corporation and also Xilinx's WebPACK ISE that is used to compile the VHDL source code and generating necessary output.

3.2 Design Method

A bottom-up design methodology starts by defining the "low level" procedures then moves up towards more and more complex procedures using those already defined. Meanwhile a top-down design methodology is the software design technique which aims to describe functionality at a very high level, then partition it repeatedly into more detailed levels one level at a time until the detail is sufficient to allow coding.

A top-down design methodology has been chosen because it is a natural way to approach a complex design task. It relies on multiple levels of abstraction to limit the number of independent concepts at each level of the design. Design is broken down into as many levels of abstraction as necessary to keep the amount of detail at each level manageable.

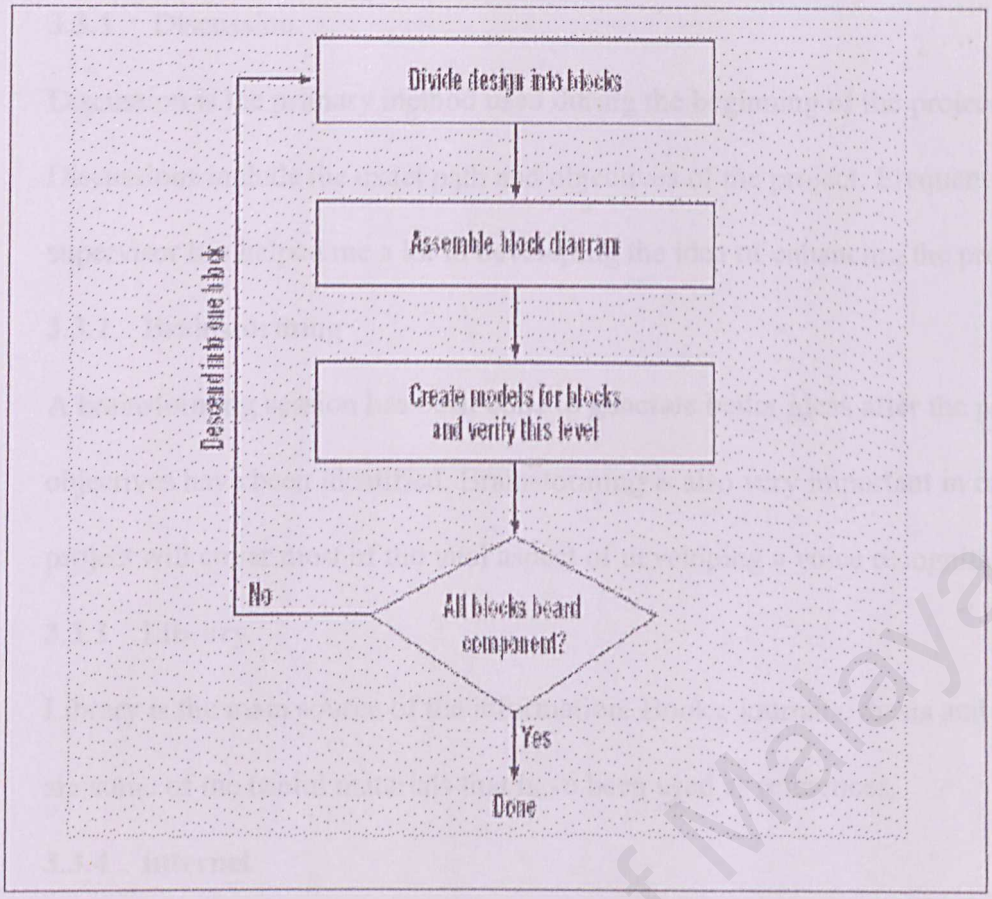


Figure 3.1 : Simplified top-down design methodology

From Figure 3.1 we can conclude that a top-down approach system is divided into blocks. Starting with the highest level of abstraction, each block is then progressively modeled, interconnected, and simulated, until the design is broken down to its most fundamental level. The benefits of top-down design methodology:

- Control over design intent
- Concurrent design collaboration
- Increased performance
- Intelligent re-use of data

3.3 Project Development Techniques

3.3.1 Discussion

Discussion is the primary method used during the beginning of the project. Discussions include the exact path and objectives of the project. Frequent visit to my supervisor has helped me a lot in developing the idea of enhancing the project.

3.3.2 Brainstorming

A brainstorming session has been done to generate better ideas after the project's objectives have been identified. Brainstorming is also very important in ensuring the project will cover most of the vital aspect of developing a voice recognition system.

3.3.3 Library

Library is the main source of the information. Books, journals, thesis and magazines are some of the useful materials that have been used as references.

3.3.4 Internet

Undeniable, Internet has become part of our life. It has become one of the main and fastest sources for information. Accessible from anywhere, information can be obtained easily from the Internet. However, validation has been made to ensure the reliability of the information.

3.4 Development Tools

3.4.1 XStend Board V1.3.2 and XS 100 Board

XStend Board V1.3.2 (**Figure 3.2**) and XS 100 Board (**Figure 3.3**) are chosen as the development tools of this project. The XStend Board contains resources such as the pushbuttons, DIP switches, LEDs, and prototyping area that are useful for basic lab experiments. The XS 100 Board offer a flexible, low-cost method of prototyping FPGA.

However, their small physical size limits the amount of support circuitry they can hold. The XStend Board removes this limitation by providing additional support circuitry that the XS 100 can access through their breadboard interfaces.

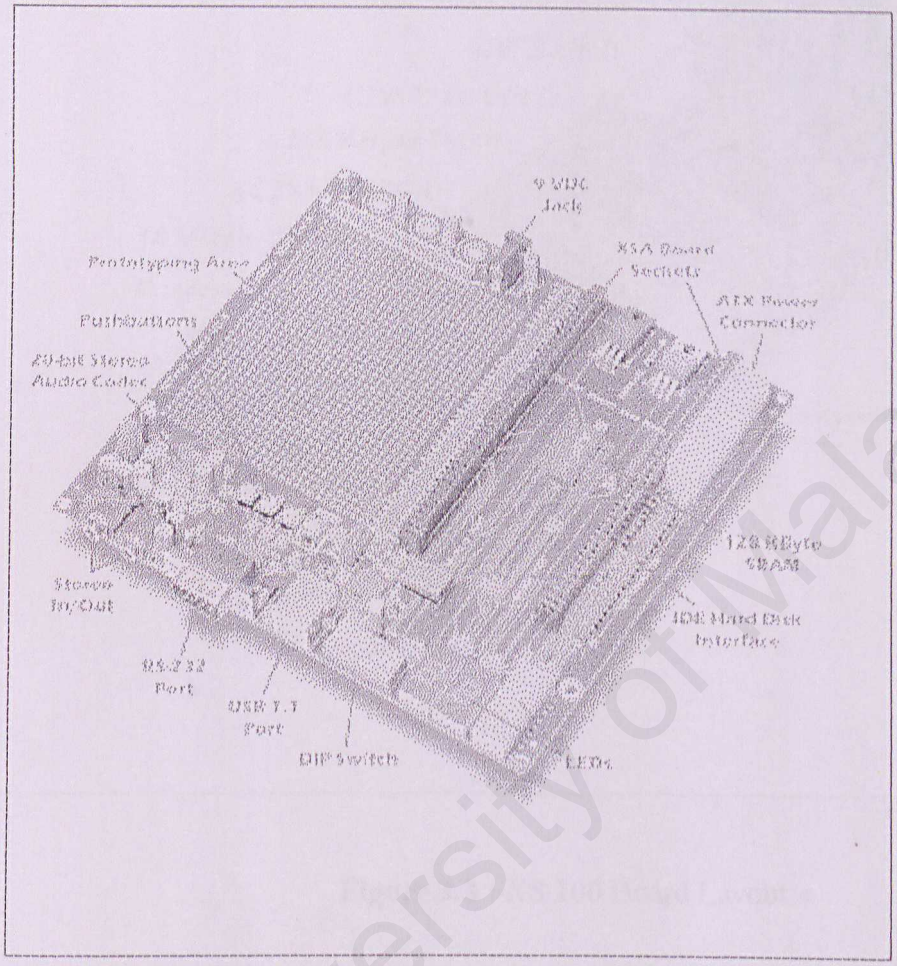


Figure 3.2 : Xstend Board Layout

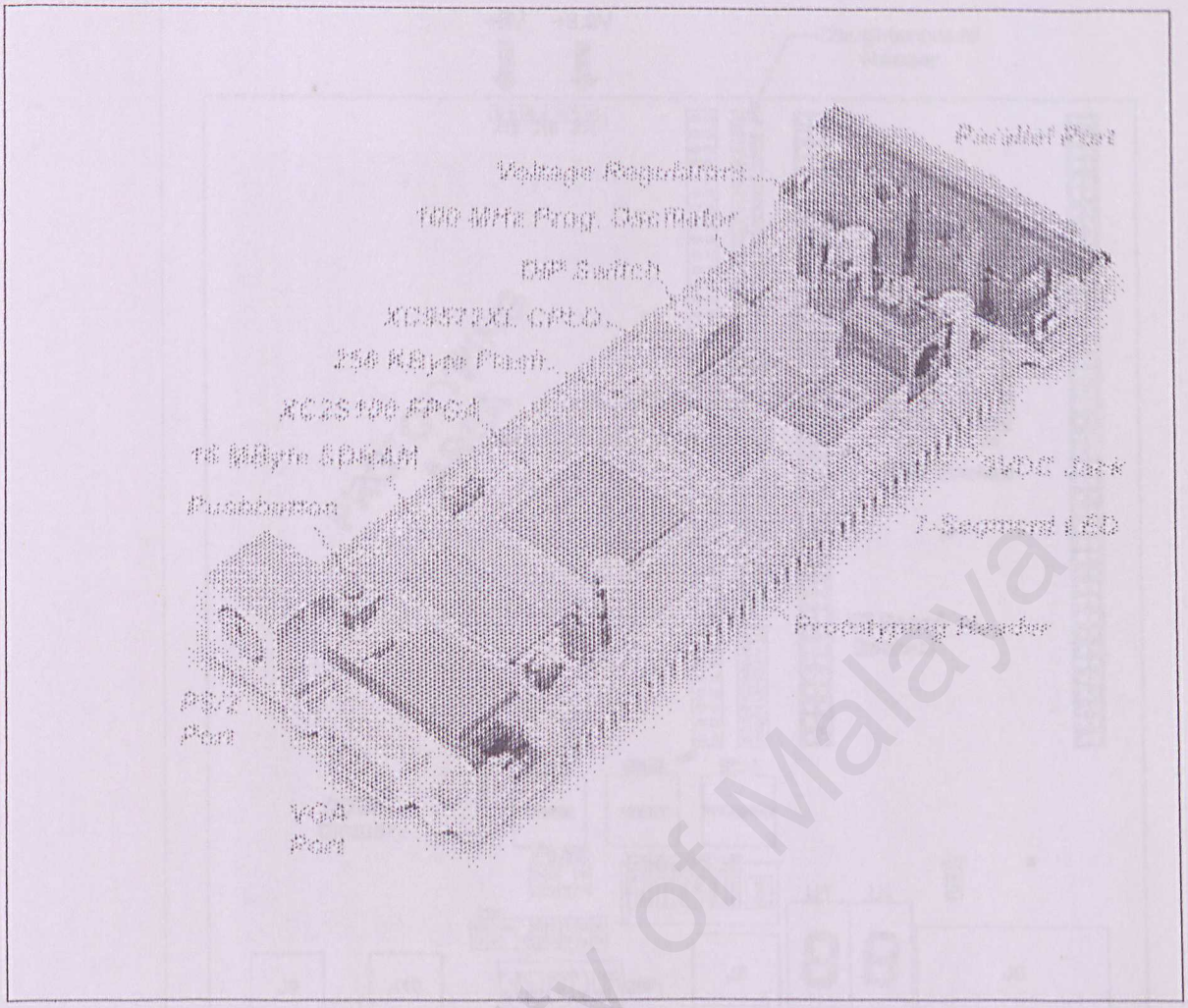


Figure 3.3 : XS 100 Board Layout

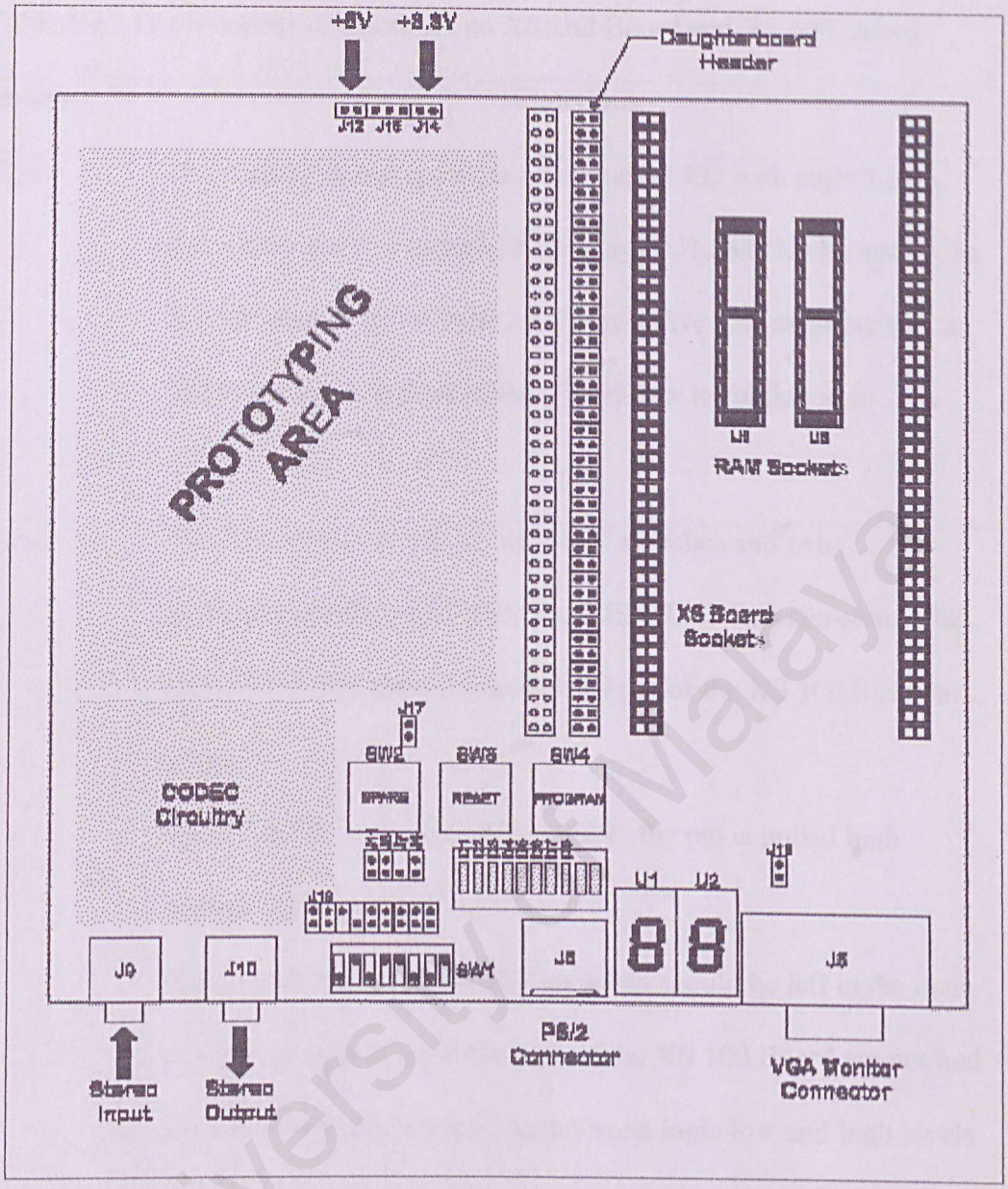


Figure 3.4 : XStend Board view

These resources are shown in the simplified view of the XStend Board and XS 100 Board (**Figure 3.4**). Each of these resources will be described next.

Table 3.1 : The functions of resources on XStend Board and XS 100 Board.

Resources	Functions
LEDs	The XStend Board provides a bar graph LED with eight LEDs (D1—D8) and two more LED displays (U1 and U2) for use by an XS 100 Board. All of these LEDs are active-low meaning that an LED segment will glow when a logic-low is applied to it.
Switches	The XStend has a bank of eight DIP switches and two pushbuttons (labeled SPARE and RESET). When closed or ON, each DIP switch pulls the connected pin of the XS 100 Board to ground. When the DIP switch is open or OFF, the pin is pulled high through a 10k resistor. When not being used, the DIP switches should be left in the open or OFF configuration so the pins of the XS 100 Board are not tied to ground and can freely move between logic low and high levels. When pressed, each pushbutton pulls the connected pin of the XS 100 Board to ground. Otherwise, the pin is pulled high through a 10k resistor.
VGA Interface	The XStend Board provides an XS 100 Board with an interface to a VGA monitor through connector J5.

PS/2 Keyboard Interface	It provides a clock signal and a serial data stream that is synchronized with the falling edges on the clock signal.
RAMs	The chip-selects of the XStend Board RAMs are connected to different pins so all the RAMs can be individually selected.
Stereo Codec	Accepts two analog input channels from J9, digitizes the analog values, and sends the digital values to the XS 100 Board as a serial bit stream. The codec also accepts a serial bit stream from the XS 100 Board and converts it into two analog output signals, which exit the XStend Board through J10.

3.4.2 XILINX WebPACK ISE

The software that is chosen to support the programmable logic deign used in voice recognition is Xilinx's WebPACK ISE (**Figure 3.5**). Xilinx is the world's leading innovator of complete programmable logic solutions. WebPACK ISE is a software solution that contains support for advanced HDL entry, synthesis, simulation, and verification capabilities for both CPLD and FPGA designs. (**Figure 3.6 and Figure 3.7**)WebPACK ISE modules provide complete design implementation control.

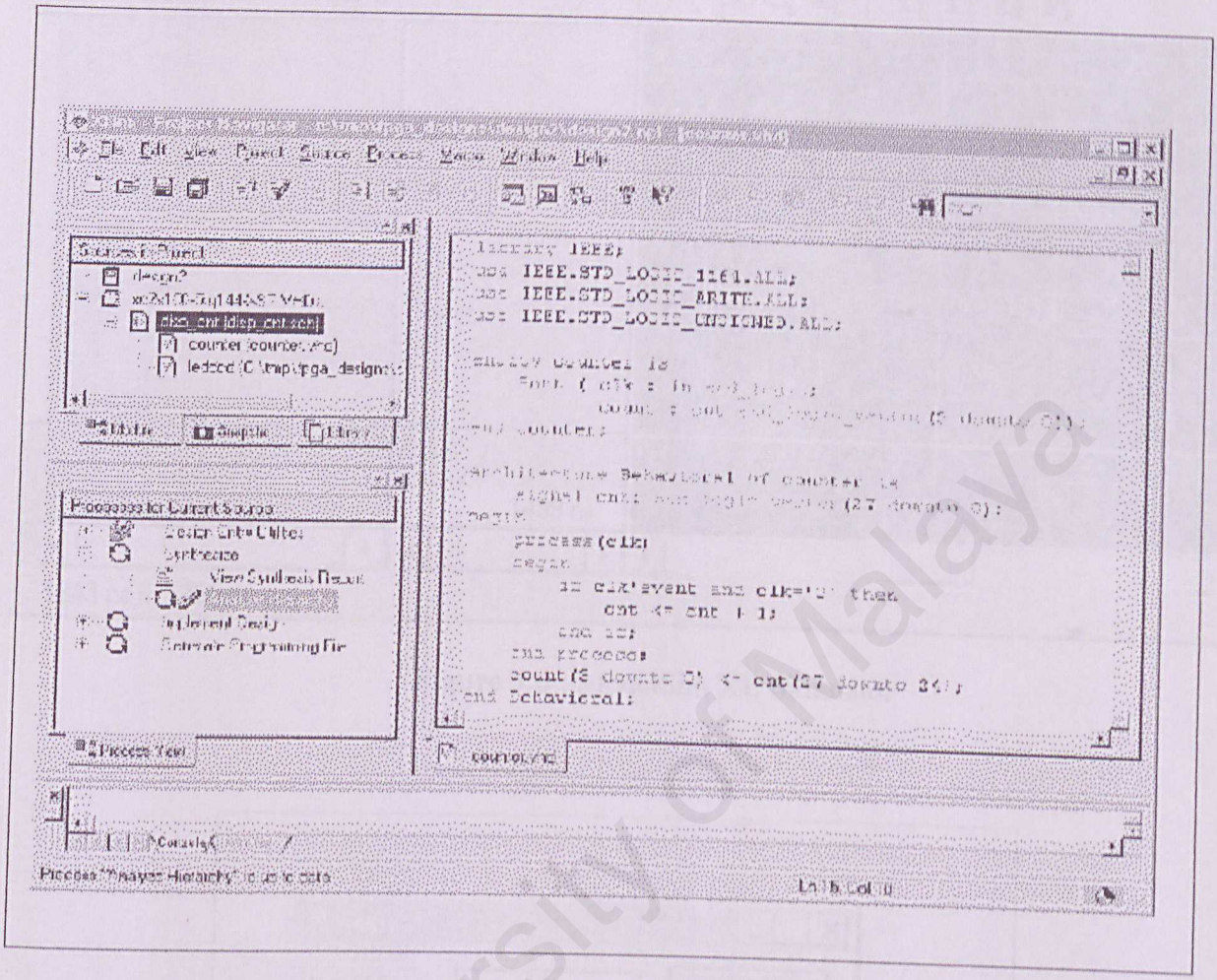


Figure 3.5 : Xilinx WebPack 4.2

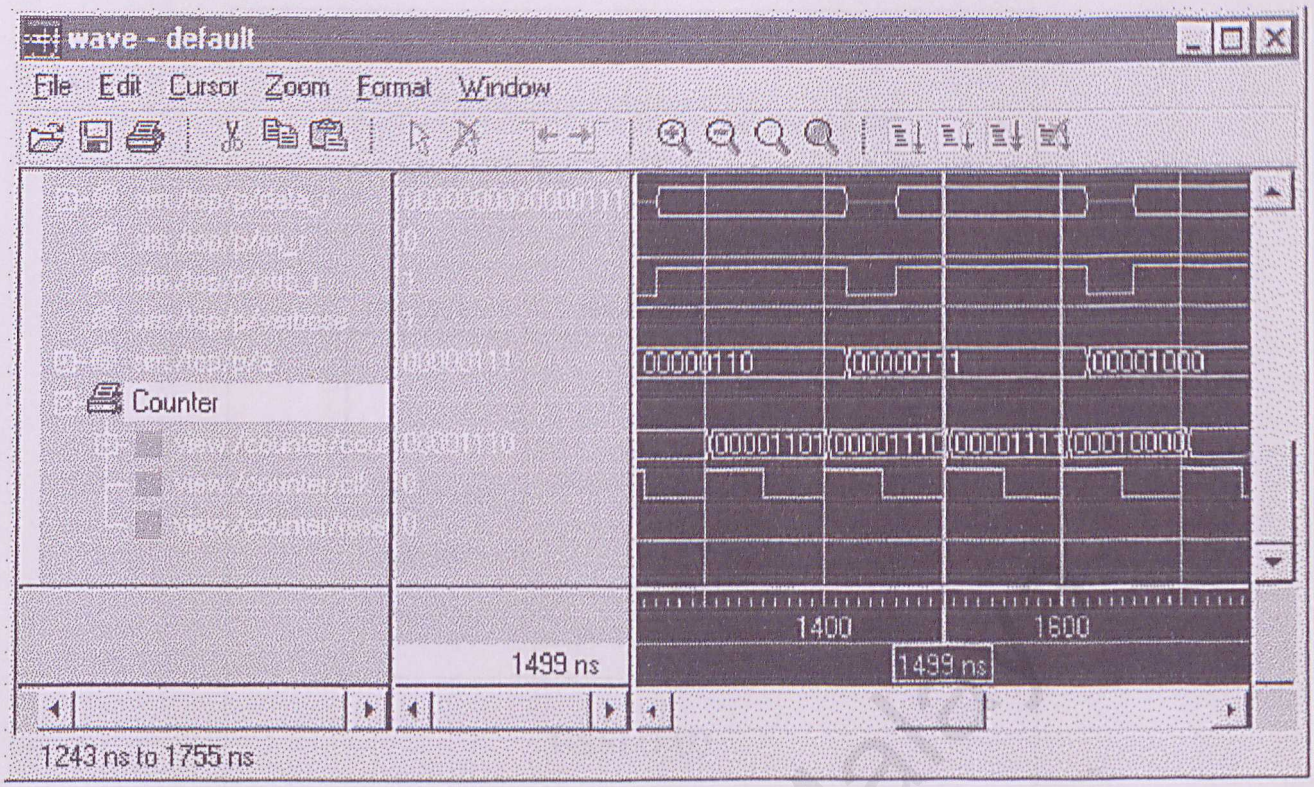
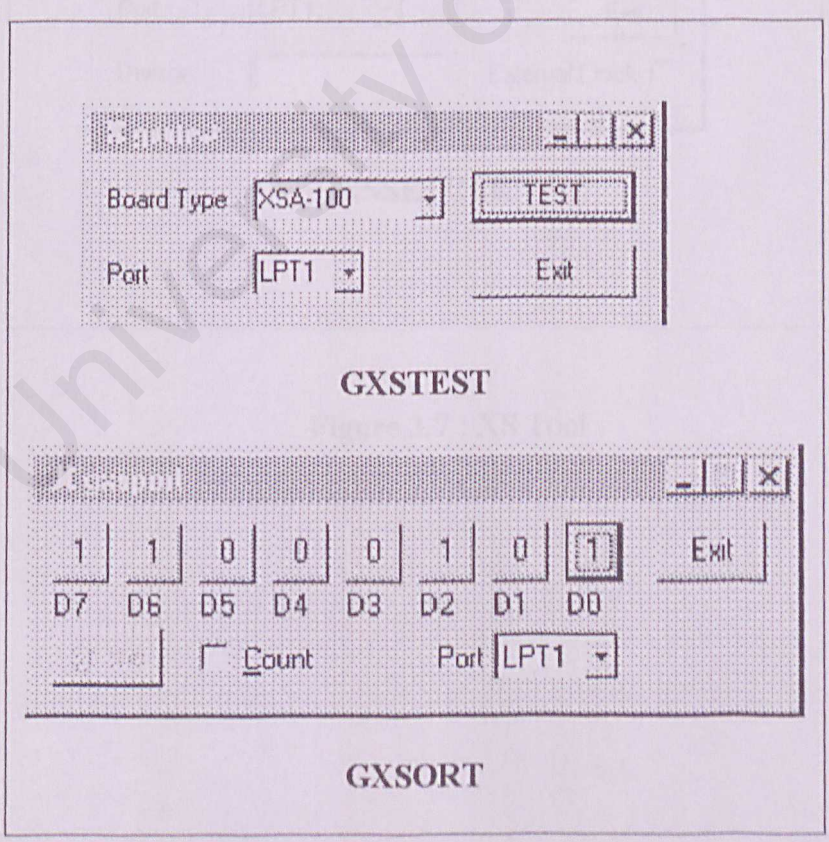


Figure 3.6 : ModelSim XE Simulator



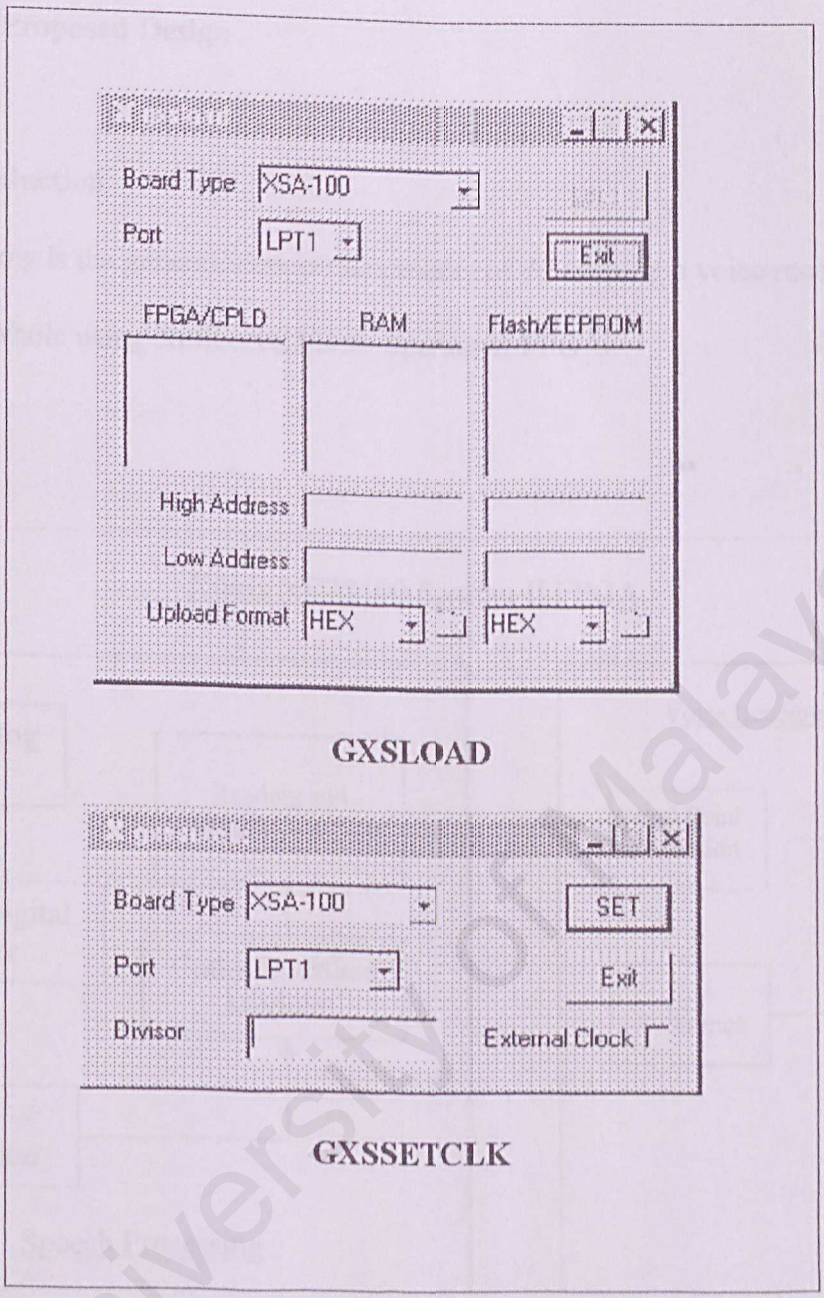


Figure 3.7 : XS Tool

Chapter 4: Proposed Design

4.1 Introduction

Provided below is the general view of the process of developing a voice recognition system as a whole using Xilinx XC2S100 Spartan-II FPGA.

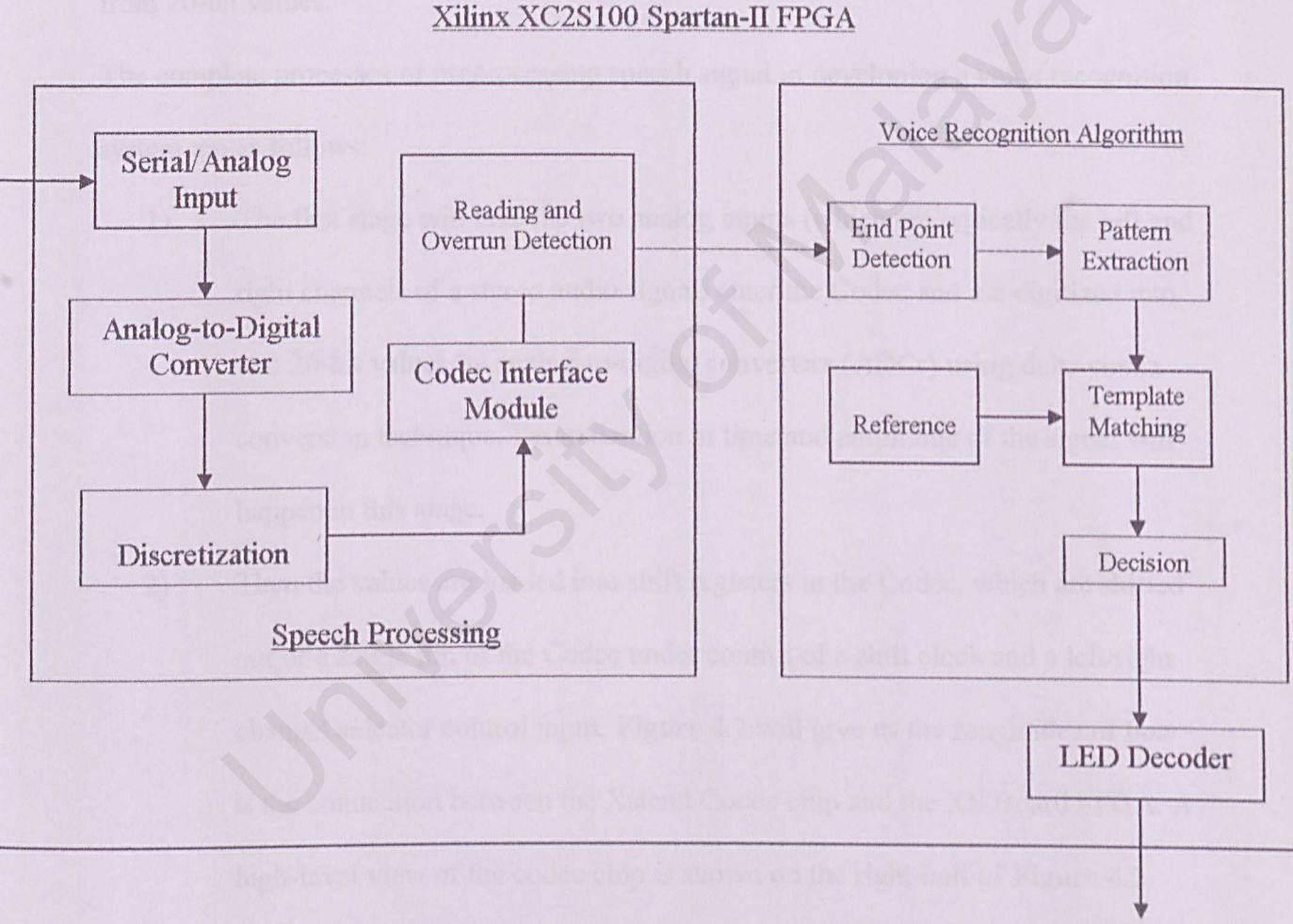


Figure 4.1 : The general view of the voice recognition process

4.2 Inputting Voice Signals through the Xstend Board V 1.3.2 Codec and outputting it to the LED

The Preprocessing circuit will take a stereo input and output it through a voice recognition algorithm to the LED bars. The pre-processing block is used to adapt the characteristics of the input signal to the recognition system. The stereo Codec on the XStend Board (a product from Cirrus Logic, CS4222) is capable of digitizing two analog signals to 20 bits of resolution while simultaneously generating two analog signals from 20-bit values.

The complete processes of preprocessing speech signal in developing a voice recognition system are as follows:

1) The first stage will take the two analog inputs (which are typically the left and right channels of a stereo audio signal) enter the Codec and are digitized into two 20-bit values by analog-to-digital converters (ADCs) using delta sigma conversion technique. Discretization in time and amplitude of the signal will happen in this stage.

2) Then the values are loaded into shift registers in the Codec, which are shifted out of a single pin of the Codec under control of a shift clock and a left/right channel selector control input. **Figure 4.2** will give us the rough idea of how is the connection between the Xstend Codec chip and the XS Board FPGA. A high-level view of the codec chip is shown on the right-half of **Figure 4.2.**

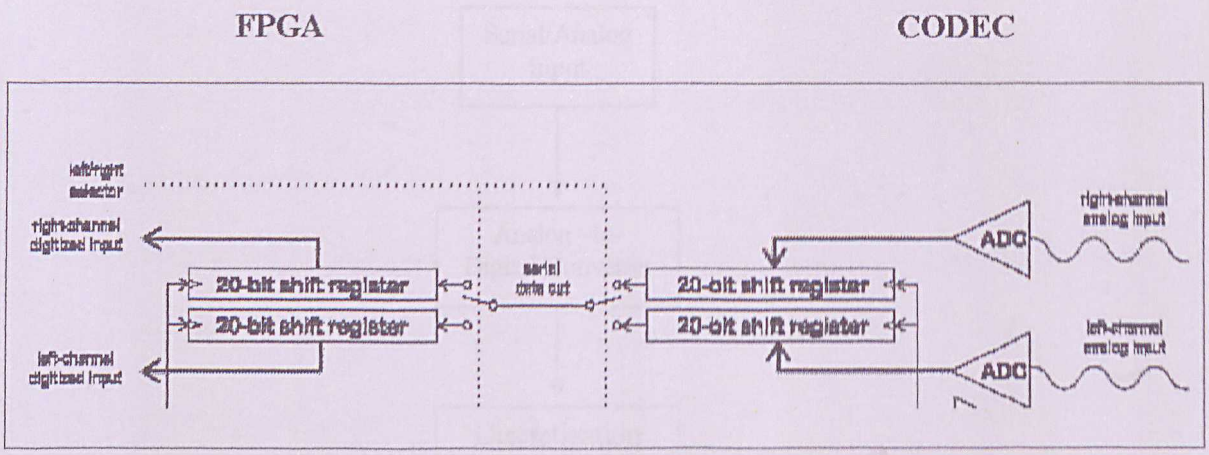


Figure 4.2: Connections between XStend Codec chip and FPGA

3) From **Figure 4.2**, we can see that the FPGA contain a set of shift register. This serial-in parallel out shift register will convert the serial input stream into 20 bits values.

4) Reading operation will be performed once the output is generated. Overflow of the FPGA shift register will also be detected here if they are not read in time.

5) The speech or voice signal will be analyzed by the chosen voice recognition algorithm to separate the phoneme sounds to produce a detailed phoneme representation of the utterance.

6) After the specific algorithm has been performed on the voice signal, the voice signal will be sent to a 20- bit register (which is actually a latch) that will store the signal before it is sent to the LED decoder.

7) Finally, LED will display the numbers uttered by the speaker, on the LED bars. **Figure 4.3** will show the block diagram of the entire process from inputting the voice signals through the Codec and then outputting it to the LED bar.

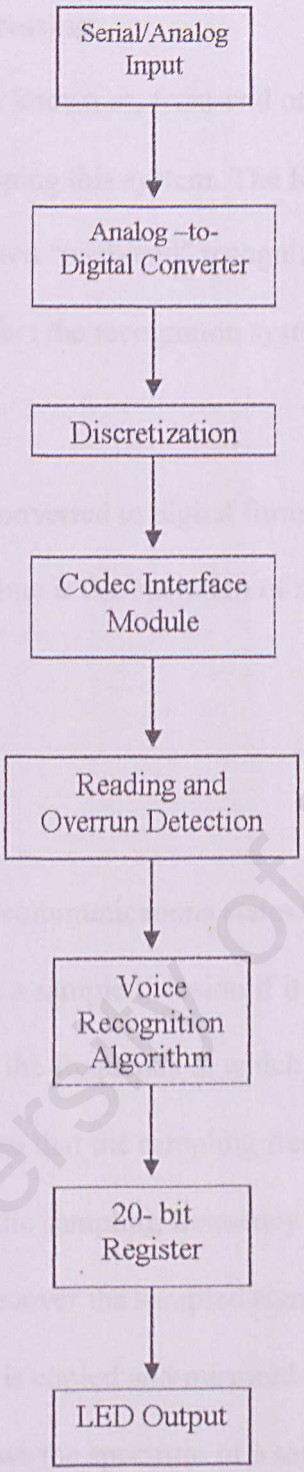


Figure 4.3 : The block diagram of the entire process

4.3 Voice Signal Preprocessing

Voice signal preprocessing or known as, front-end of the speech recognition system is the most important part in developing this system. The features here are transmitted over an interface channel to a completed "back-end" recognizer. The end result is that the voice transmission that does not affect the recognition system performance.

4.3.1 Speech Digitization

When an analogue signal is converted to digital form, it is made discrete both in time and amplitude. Discretization in time is the operation of *sampling*, while in amplitude it is *quantization.*

- **Sampling**

A fundamental thermo of telecommunications states that a signal can only be reconstructed accurately from a sampled version if it does not contain components whose frequency is greater than half the frequency at which the sampling takes place. Say, the sampling interval T seconds, so that the sampling frequency is 1/T Hz. [Ian, 1982]The sampling theorem states that the sampling frequency of a signal must be at least twice the signal frequency in order to recover the sampled signal without distortion. When a signal is sampled its input spectrum is copied and mirrored at multiples of the sampling frequency *fS*. **Figure 4.4** shows the spectrum of a sampled signal when the sampling frequency *fS* is less than twice the input signal frequency 2f0. The shaded area on the plot shows what is commonly referred to as *aliasing* which results when the sampling theorem is violated. Aliasing is a phenomenon where components at frequencies just under the sampling frequency masquerade as a very low-frequency components.

Recovering a signal contaminated with aliasing results in a distorted output signal. Figure 4.5 show the spectrum of an over sampled signal. The oversampling process puts the entire input bandwidth at less than fS/2 and avoids the aliasing trap. The analog signal is continuous in time and it is necessary to convert this to a flow of digital values. It is therefore required to define the rate at which new digital values are sampled from the analog signal. The rate of new values is called sampling rate. The sampling rate to be used is 46.8 KHz.(See Codec Interface Module)

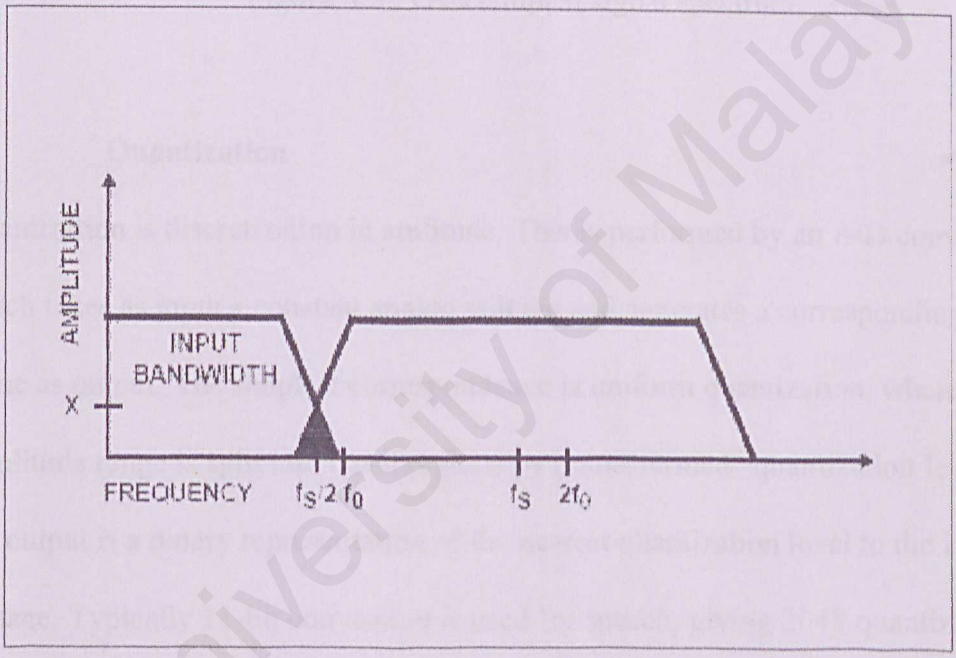


Figure 4.4: Undersampled signal spectrum

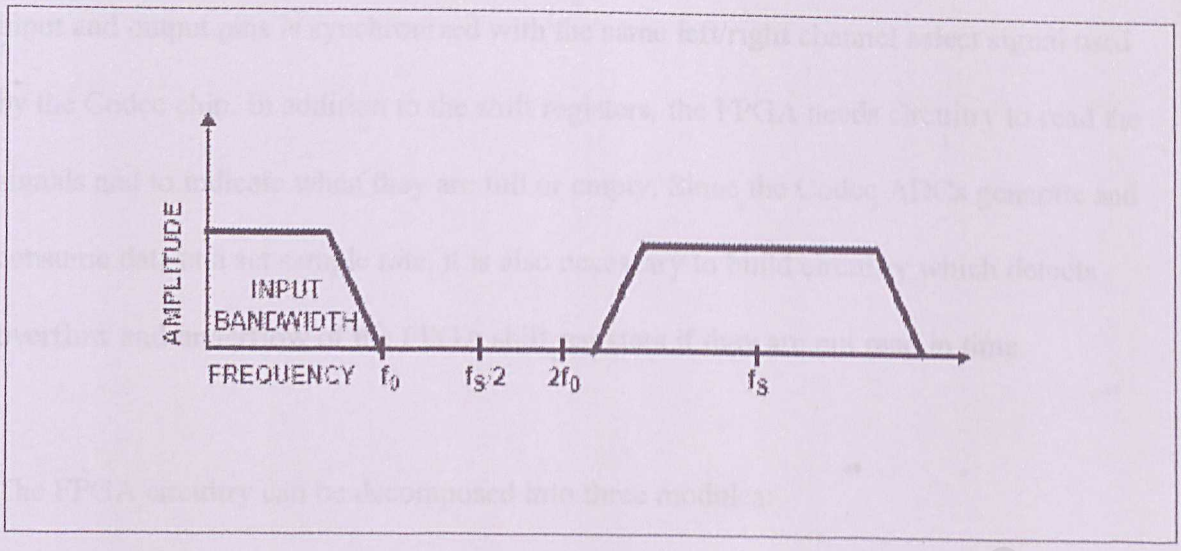


Figure 4.5 : Oversampled signal spectrum

- **Quantization**

Quantization is discretization in amlitude. This is performed by an A/D converter which takes as input a constant analog voltage and generates a corresponding binary value as output. The simplest correspondence is uniform quantization, where the amplitude range is split into equal regions by points termed "quantization levels", and the output is a binary representation of the nearest quantization level to the input voltage. Typically 11-bit conversion is used for speech, giving 2048 quantization levels, and the signal is adjusted to have zero mean so that half the levels correspond to negative input voltages and the other half to positive ones.

4.4 FPGA (Field Programmable Gate Array) Circuitry

The FPGA is handling these values in a bit-parallel manner, so the FPGA must contain a set of shift registers which convert the serial input stream into 20-bit values. This is shown in the left-half of **Figure 4.1**. The gating of these shift registers onto the serial

input and output pins is synchronized with the same left/right channel select signal used by the Codec chip. In addition to the shift registers, the FPGA needs circuitry to read the signals and to indicate when they are full or empty. Since the Codec ADCs generate and consume data at a set sample rate, it is also necessary to build circuitry which detects overflow and underflow of the FPGA shift registers if they are not read in time.

The FPGA circuitry can be decomposed into three modules:

► A clock generator module which outputs the serial data shift clock and the left/right channel select signals;

► A channel module which contains the shift registers, buffers, read/write control, and overflow/underflow detection circuitry for a single input/output stream of data;

► A top-level module, which combines the clock generator module with two channel modules to form a complete Codec, interface circuit.

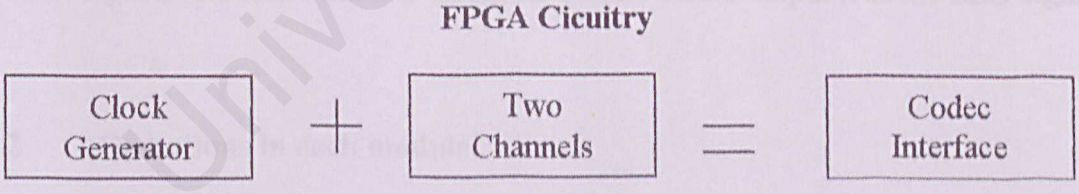


Figure 4.6: A simplified view of FLPD circuitry

4.4.1 Top level design of the system

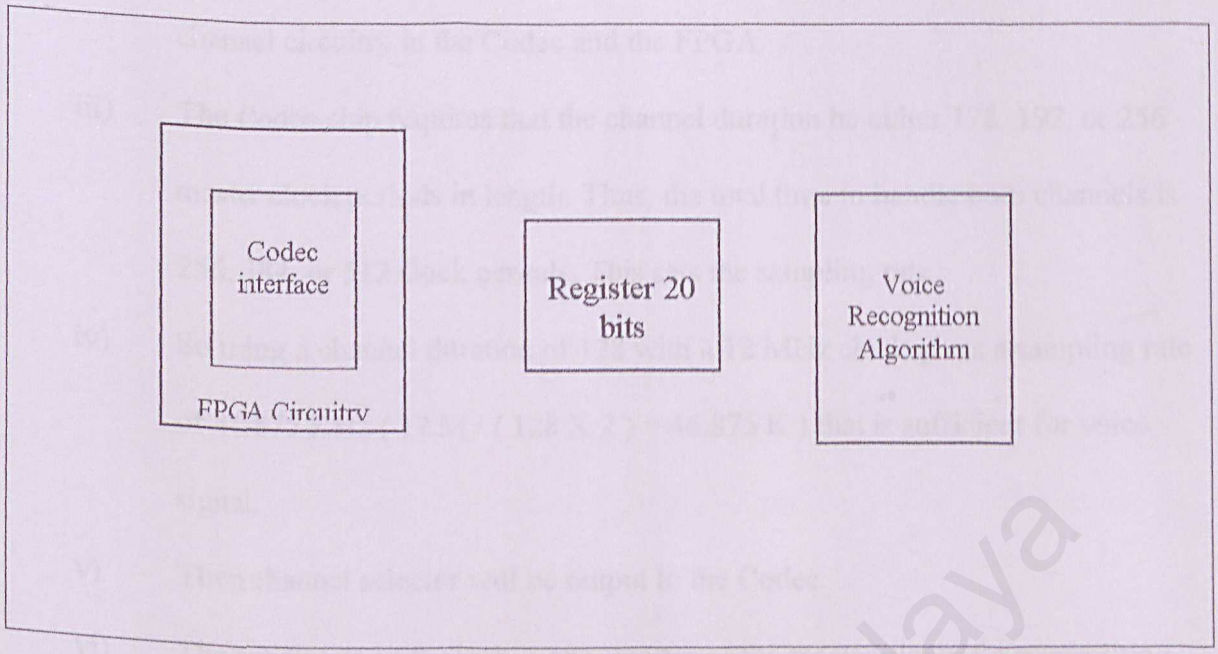


Figure 4.7 : The top level design of the Voice Recognition system

Once the Codec interface module is completed and packaged, we will use it in the voice recognition application. We will use the FPGA in accepting the left and right stereo inputs from the Codec ADCs and then perform a specific voice recognition algorithm to the voice signals and then send it to a register 20 bits before output it as the LED signals.

4.4.2 Operations in each module

- **Clock Generator**

A clock generator module is used to output the serial data shift clock and the left/right channel select signals.

i) The typically main clock input which is going to be used here is the 12 Mhz clock from XS Board.

ii) There will be an output that controls the activation of the left and right channel circuitry in the Codec and the FPGA.

iii) The Codec chip requires that the channel duration be either 128, 192, or 256 master clock periods in length. Thus, the total time to handle both channels is 256, 384, or 512 clock periods. This sets the sampling rate.

iv) So using a channel duration of 128 with a 12 MHz clock gives a sampling rate of 46.875 KHz (12 M / (128 X 2) = 46.875 K) that is sufficient for voice signal.

v) Then channel selector will be output to the Codec.

vi) The serial data shift clock is one-quarter of the master clock. So transmitting or receiving a 20-bit value will require 4 X 20 =80 clock periods, and this will fit within the shortest possible channel duration.

vii) Finally a process of incrementing the sequencing counter and toggling the left/right channel selector will be performed when the count reaches the duration for which a channel is active.

- **Channel module**

It contains a shift registers, buffers, read control, and overflow/underflow detection circuitry for a single input/output stream of the data.

i) There will a process of receiving serial data stream from the Codec ADC that is shifted in, through an sdout input.

ii) A shift register and a flag will both be used in indicating the current status of the shift register.

iii) The status of the shift register will change whenever it is accepting serial data.

iv) The shift register status changes to *full* as soon as the last bit enters the shift register.

v) A flag is maintained that indicates whether the contents of the ADC shift register have been read. The flag is set when the ADC register for the channel is full and it is selected for a read operation. The flag will stay set after the read operation is complete.

vi) There will also be a process of monitoring and detecting an error condition of the ADC shift register and flags. This happen when the register begins accepting bits from the current sample period but the data from the previous period has not yet been read and causes the process overwriting of data from the Codec ADC channels. This is known as *overrun* error.

- **Read operation**

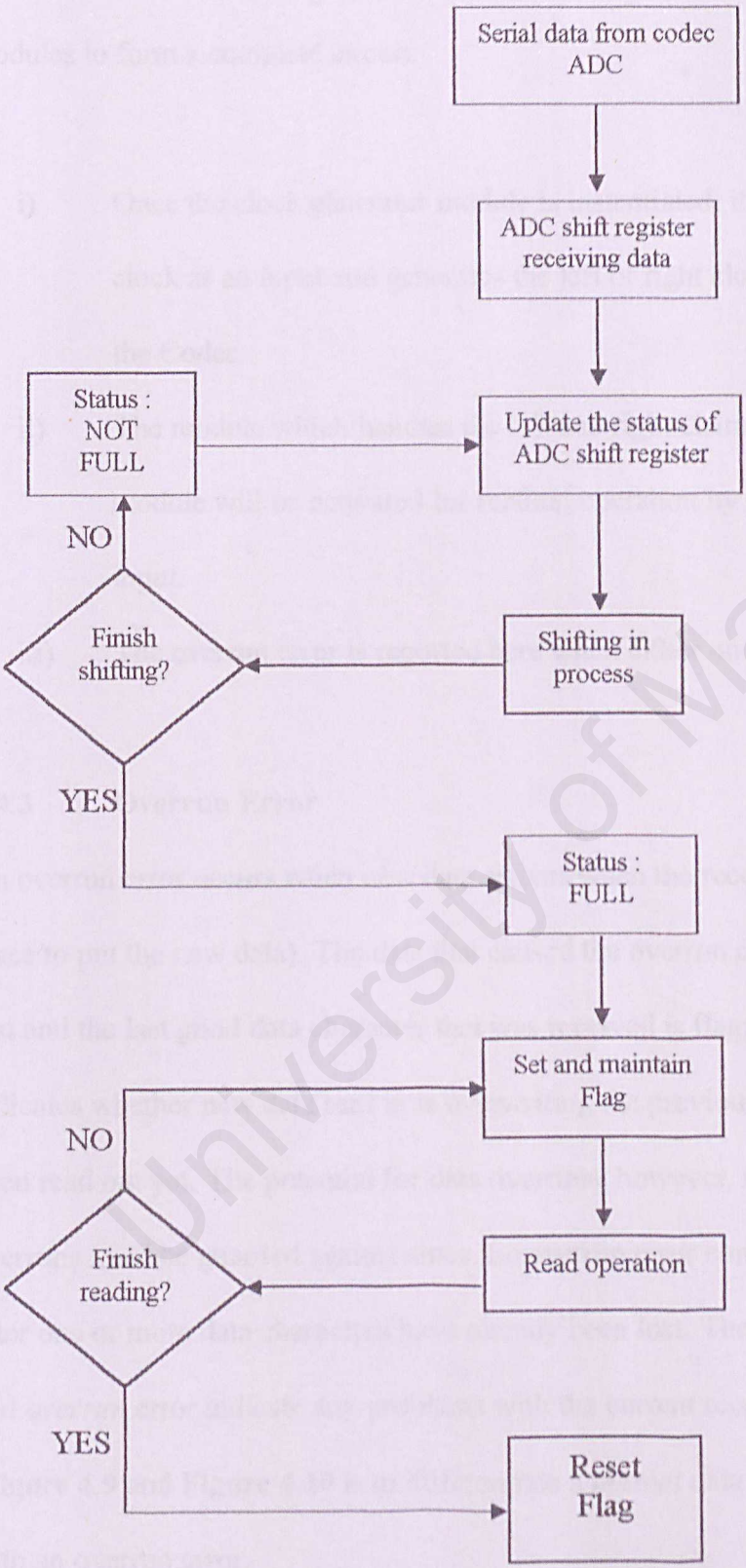


Figure 4.8 : Block diagram of the reading operation

- **Codec Interface**

It is a top-level module, which combines the clock generator module with two channel modules to form a complete circuit.

i) Once the clock generator module is instantiated, it will receive the 12 MHz clock as an input and generates the left or right clock and serial shift clock for the Codec.

ii) The module which handles the left and right channels is instantiated and this module will be activated for reading operation by the left or right selection input.

iii) The overrun error is reported here when either one channel reports an error

4.4.3 Overrun Error

An overrun error occurs when new data arrives when the receive buffer is full (there's no place to put the new data). The data that caused the overrun condition to be detected is lost and the last good data character that was received is flagged with the overrun error. It indicates whether new data sent in is overwriting the previous data received that has not been read out yet. The potential for data overruns, however, is always present. Data overruns must be guarded against since the overrun error condition can only be detected after one or more data characters have already been lost. The *parity error*, *framing error*, and *overrun* error indicate any problems with the current received data.

Figure 4.9 and **Figure 4.10** is to differentiate a normal data flow in a register and a flow with an overrun error.

a) The normal data flow in a register

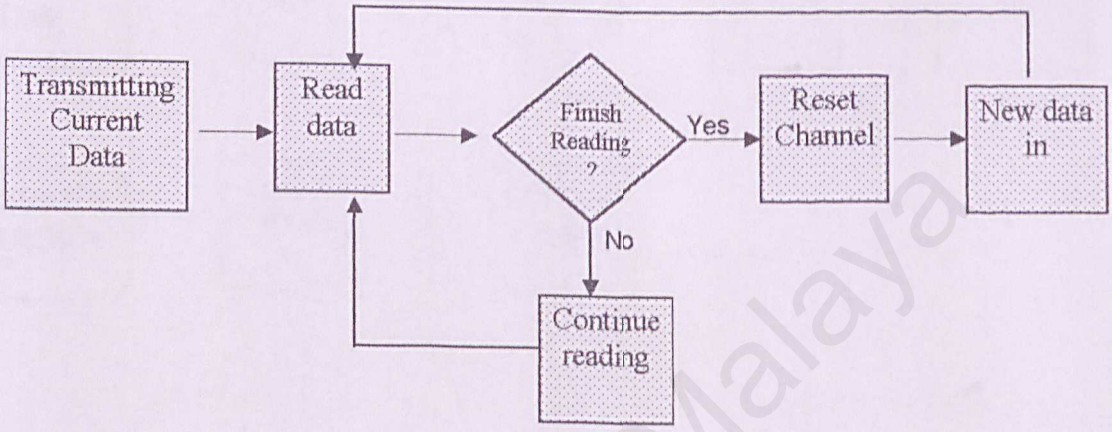


Figure 4.9 : Normal Flow

b) Overrun error / Overflow in register

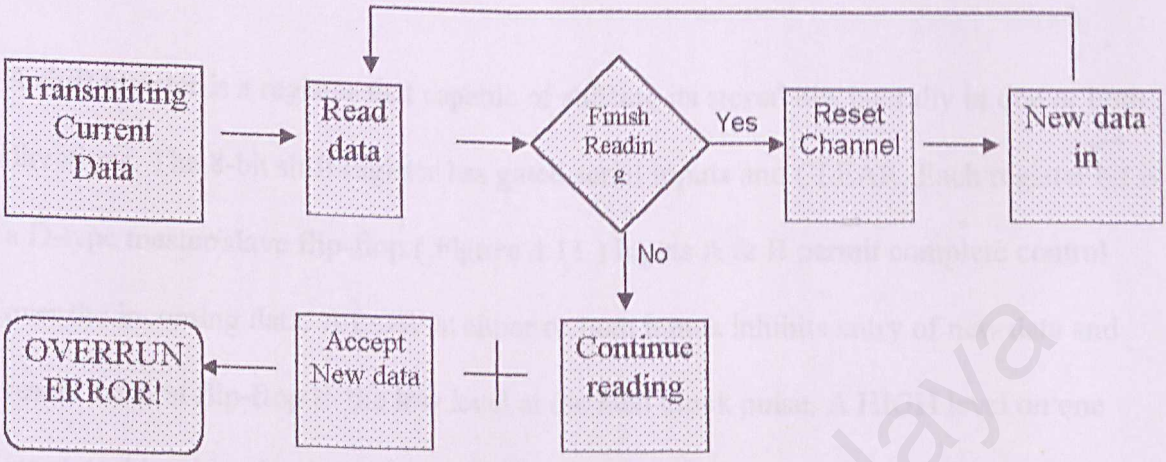


Figure 4.10 : Register with overrun error

4.5 Shift Register

There are only two basic ways for sending and receiving digital data. These methods are known as parallel transmission and serial transmission. In parallel transmission, all the bits that make up a byte of data are sent at one time. The bits are all lined up in parallel. The other method for sending data from one device to another is serial transmission. Serial transmission consists of sending the bits out one at a time.

The FPGA is handling these values in a bit-parallel manner, so the FPGA must contain a set of shift registers which convert the serial input stream into 20-bit values.

A shift register is a register that capable of shifting its stored bits laterally in one or both directions. The 8-bit shift register has gated serial inputs and CLEAR. Each register bit is a D-type master/slave flip-flop.(**Figure 4.11**) Inputs A & B permit complete control over the incoming data. A LOW at either or both inputs inhibits entry of new data and resets the first flip-flop to the low level at the next clock pulse. A HIGH level on one input enables the other input which will then determine the state of the first flip-flop. Voice signal at the serial inputs may be changed while the clock is HIGH or LOW, but only information meeting the setup and hold time requirements will be entered. Data is serially shifted in and out of the 8-bit register during the positive going transition of the clock pulse. Clear is independent of the clock and accomplished by a low level at the CLEAR input

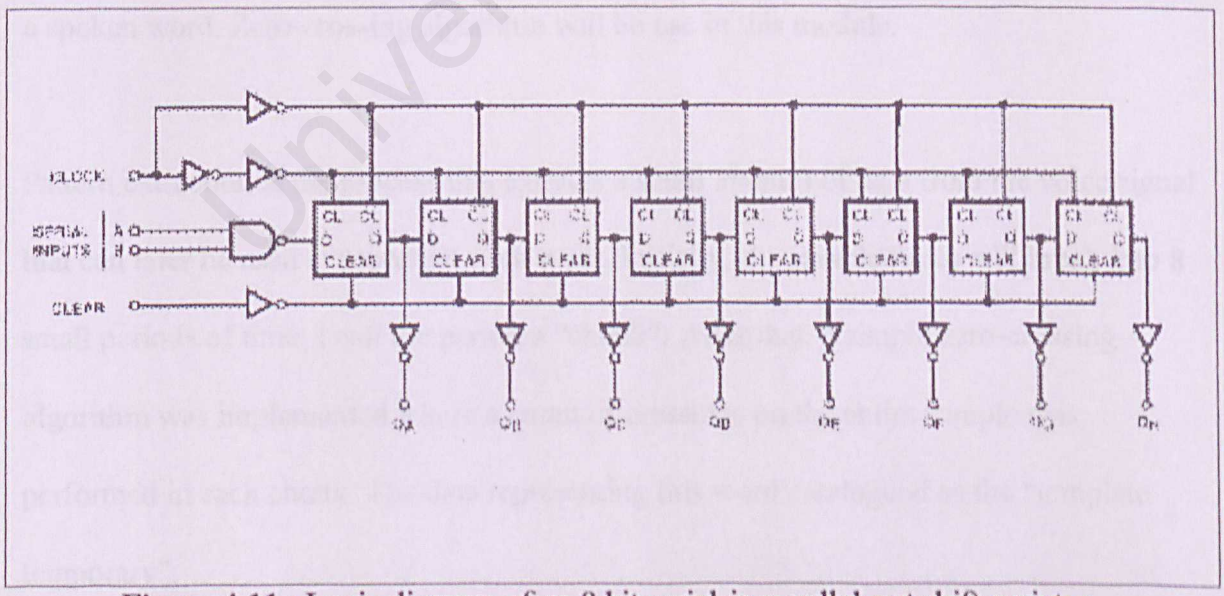


Figure 4.11 : Logic diagram of an 8 bit serial-in parallel-out shift register

4.6 Voice recognition algorithm

The voice recognition system consists of five major modules, the speech processing module, the end point module, the pattern module, pattern matching module and the decision module.

Initially, the voice data is acquired using an external microphone. Voice input is then sampled at a rate of approximately 46 kHz and with 20 bits resolution and digitizes the voice at 22050 samples per second. Of these samples, each sample is converted to a 16 bit digital representation.

Once the signal is sampled, the digitized voice is then fed into a field-programmable gate array (FPGA), the Xilinx XC2S100 Spartan-II FPGA. The End Point Detector (EDP) block use to detect the beginning and end of the word pronounced by the user. A major cause of errors in voice recognition is the inaccurate detection of beginning and ending of a spoken word. Zero-crossing algorithm will be use in this module.

Pattern extraction is the process that extracts a small amount of data from the voice signal that can later be used to represent each word. Initially, the sampled data will break into 8 small periods of time; I call the period a "chunk". After that, a simple zero-crossing algorithm was implemented where a count of crossings on the entire sample was performed in each chunk. The data representing this word catalogued as the "template temporary".

Pattern Matching involve the actual procedure to identify the unknown word by comparing extracted features from his/her voice input. In this module, the differences number of crossings between each related chunk template temporary with the templates permanent is found, and then all the differences are summed (Sum of Absolute Differences). At the same time, the differences between each related chunk is found and squared, and then the total differences are summed (Sum of Squared Differences). The final step is get the difference between Sum of Squared Differences with Sum of Absolute Differences

The decision algorithm compared the results from the pattern matching. Initially, the algorithm will find the minimum value between the results. The minimum value is then compared against a threshold value. If the minimum value is less than the threshold, then that word is chosen as the recognized word otherwise the incoming word is deemed invalid and ignored.

4.7 20- bit Register

All the bits of the register are loaded simultaneously with a common clock pulse. The symbol provided below permits the use of the register in a design hierarchy. It has all of the inputs to the FPGA circuit on its left and all of the outputs on the right. The inputs include the clock input with the dynamic indicator to represent positive-edge triggering of the flip-flops. Note that the name *clear* appears inside the symbol, with a bubble in the signal line on the outside of the symbol. This notation indicates that application of the

logic 0 to the signal line activates the clear operation on the register flip-flop. (Figure 4.12)

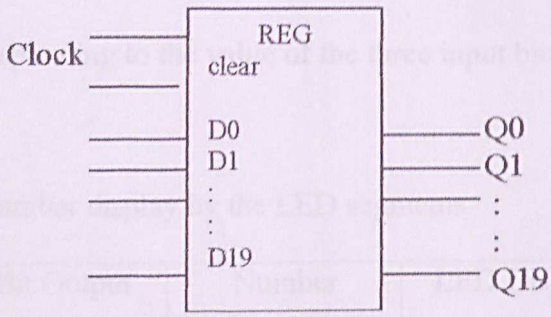


Figure 4.12 : A symbol of a 20 bits register

4.8 LED Decoder

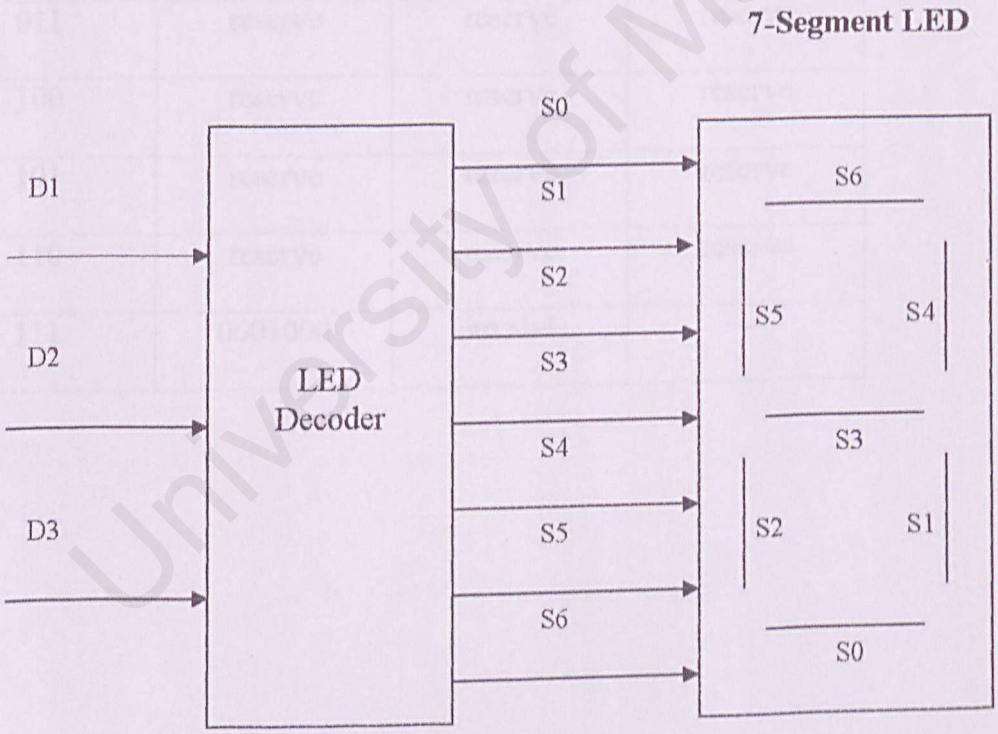


Figure 4.13 : High Level Diagram of the LED Decoder

An LED decoder takes a three bit input from the Decision Module and outputs seven signals which drive the segments of an LED digit. The LED segments will be driven to display the digit corresponding to the value of the three input bits as follows:

Table 4.1 : Number display by the LED segments

3 Bit Input	7 Bit Output	Number	LED Display
000	1110111	0	0
001	0010010	1	1
010	1011101	2	2
011	reserve	reserve	reserve
100	reserve	reserve	reserve
101	reserve	reserve	reserve
110	reserve	reserve	reserve
111	0001000	invalid	-

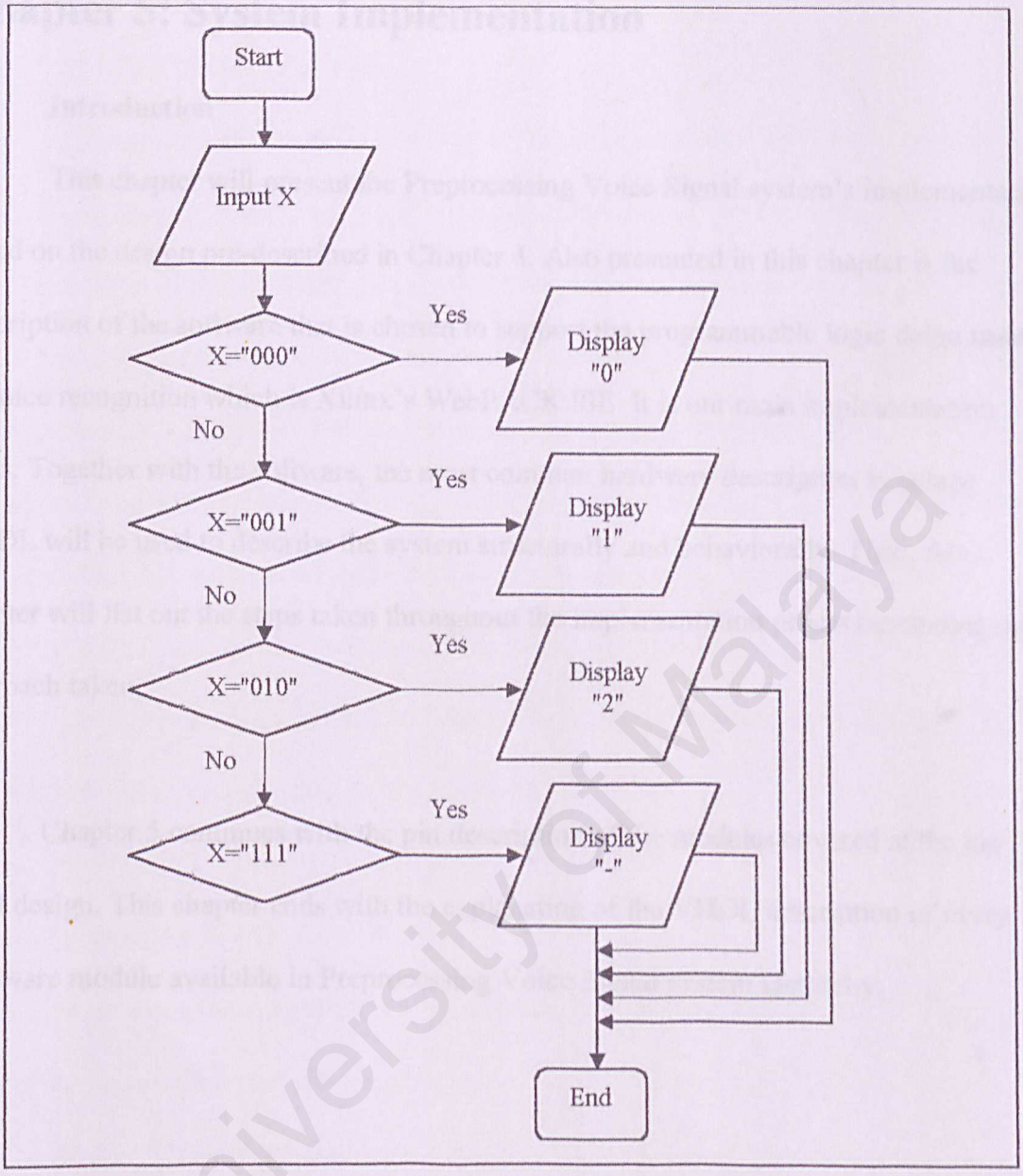


Figure 4.14 : LED Decoder Flow Chart

Chapter 5: System Implementation

5.1 Introduction

This chapter will present the Preprocessing Voice Signal system's implementation based on the design pre-described in Chapter 4. Also presented in this chapter is the description of the software that is chosen to support the programmable logic deign used in voice recognition which is Xilinx's WebPACK ISE. It is our main implementation tools. Together with the software, the most common hardware description language VHDL will be used to describe the system structurally and behaviorally. Then, this chapter will list out the steps taken throughout the implementation process including the approach taken.

Chapter 5 continues with the pin description of the modules covered at the top level design. This chapter ends with the explanation of the VHDL description of every hardware module available in Preprocessing Voice Signal system hierarchy.

5.2 XILINX WebPACK ISE

The software that is chosen to support the programmable logic deign used in voice recognition is Xilinx, the world's leading innovator of complete programmable logic solutions. WebPACK ISE is a software solution that contains support for advanced HDL entry, synthesis, simulation, and verification capabilities for both CPLD and FPGA designs. WebPACK ISE modules provide complete design implementation control. It is advanced software that integrates with VHDL for describing digital designs through an integrated VHDL simulator, Model Sim XE that comes with the product. Xilinx's WebPACK ISE can also be used in combination with other tools including schematic editors, synthesis software, high-level design tools and other tools available from third parties to form a complete design environment.

Xilinx's WebPACK ISE

It provides many useful features to help crate, modify and process VHDL projects. Some of the main features included:-

- Hierarchy browser
  - shows an up-to-date view of a design structure.
  - useful for projects involving multiple VHDL source file called modules and/or multiple levels of hierarchy.
- Module and test bench wizards
  - helps crate a new VHDL design descriptions by first asking a series of questions about the design requirements followed by the

Wizard that generates VHDL source file templates based on the requirement pre-defined.

- Built-in dependency feature
  - helps to streamline the processing of a design for simulation and synthesis
  - eliminate the need to compile each VHDL source file in a design or to keep track of source file dependencies.

The interface of Xilinx's WebPACK ISE consists of menu bar options which users click on to perform the required action. Using Xilinx's WebPACK ISE, users can either create entirely new projects or manage existing VHDL projects. In Xilinx's WebPACK ISE, the design flow starts with creating new project or opening an existing project (Figure 5.1). The browser will list out the VHDL source files. Double clicking on any of the file listed will open the editing window for that particular source file (Figure 5.2) . The next step is to compile the source file or in other words is to check for any errors occurs in the source code. Each of the compiled source files will be linked together and an executable simulation file will be generated. Project can be synthesized, to be implemented in design and generating programming file.

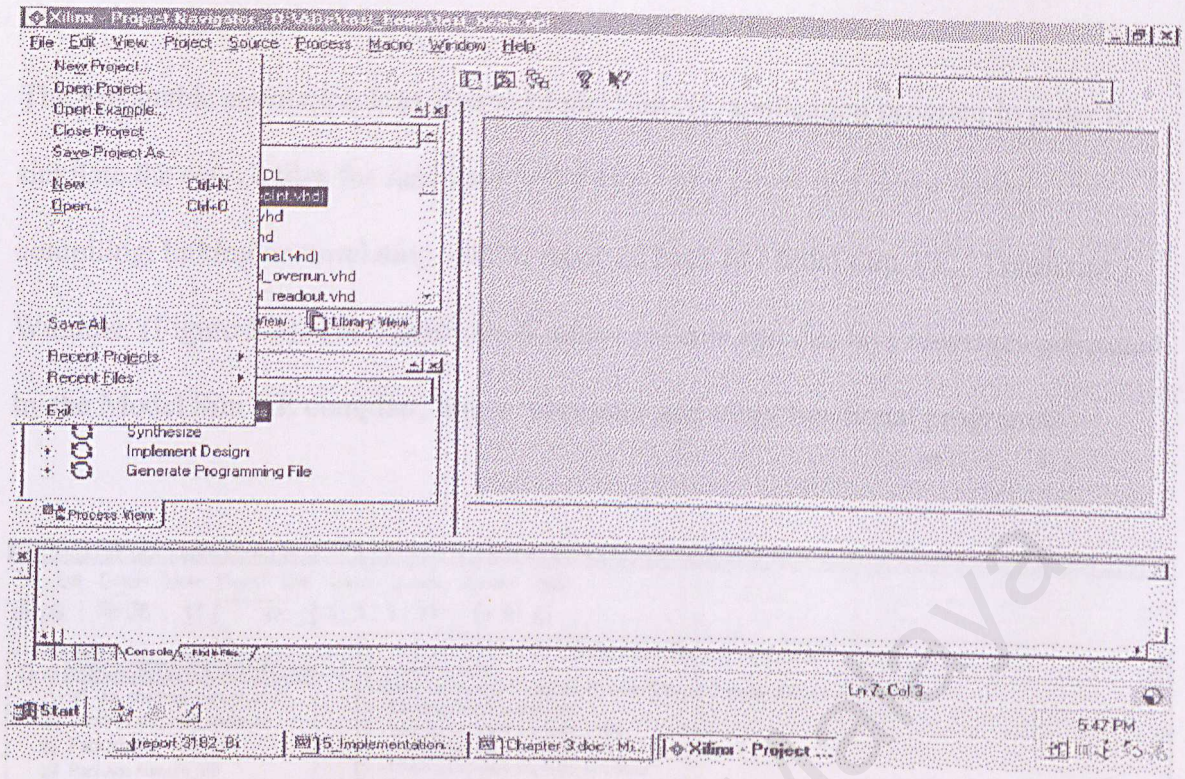


Figure 5.1: Xilinx's WebPACK ISE interface on 'File' menu bar options

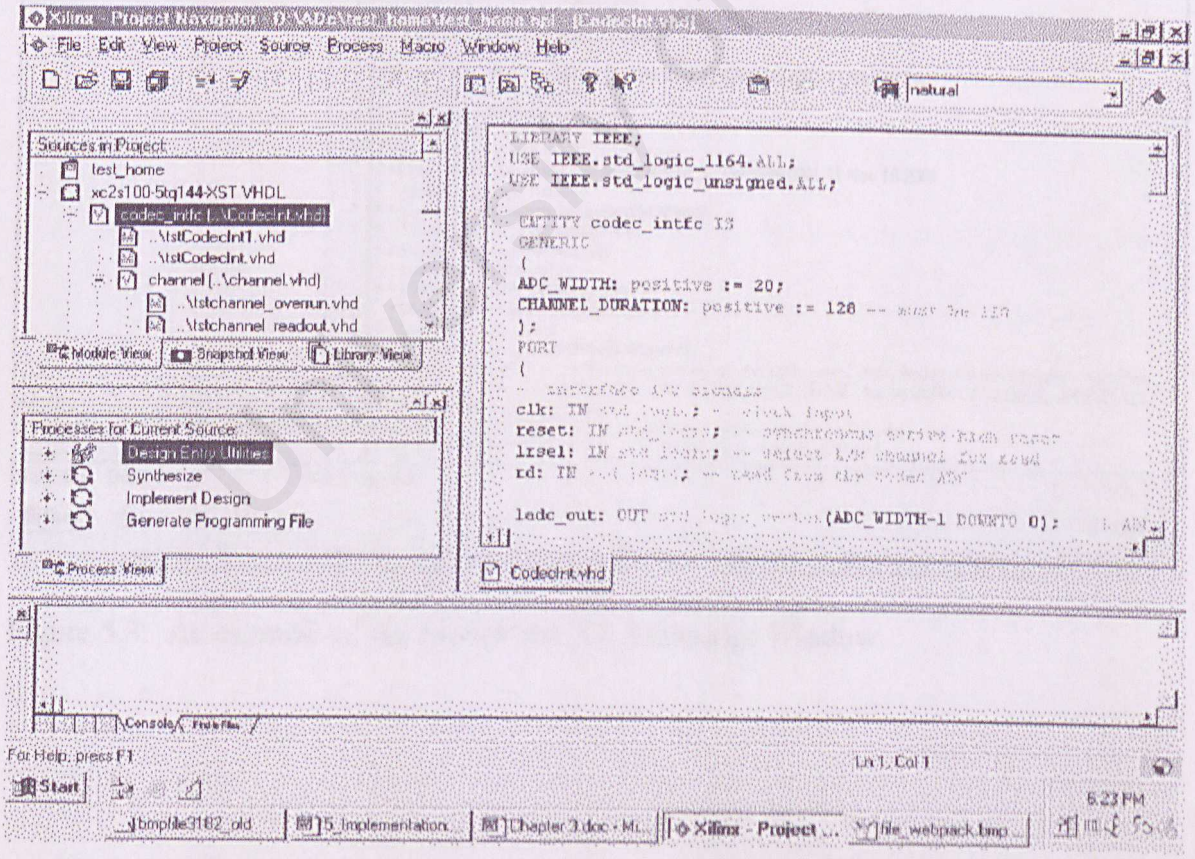


Figure 5.2: Xilinx's WebPACK ISE interface with the editing window

Model Sim XE

The executable simulation is loaded to Model Sim XE application program in order to simulate the source files for functional validation. Any errors that occur during compiling, linking or simulating will be reported back to Model Sim XE 'Transcript Window' (Figure 5.3). The Transcript Window collects and displays messages generated by the Model Sim XE compiler, linker, simulator and other functional programs.

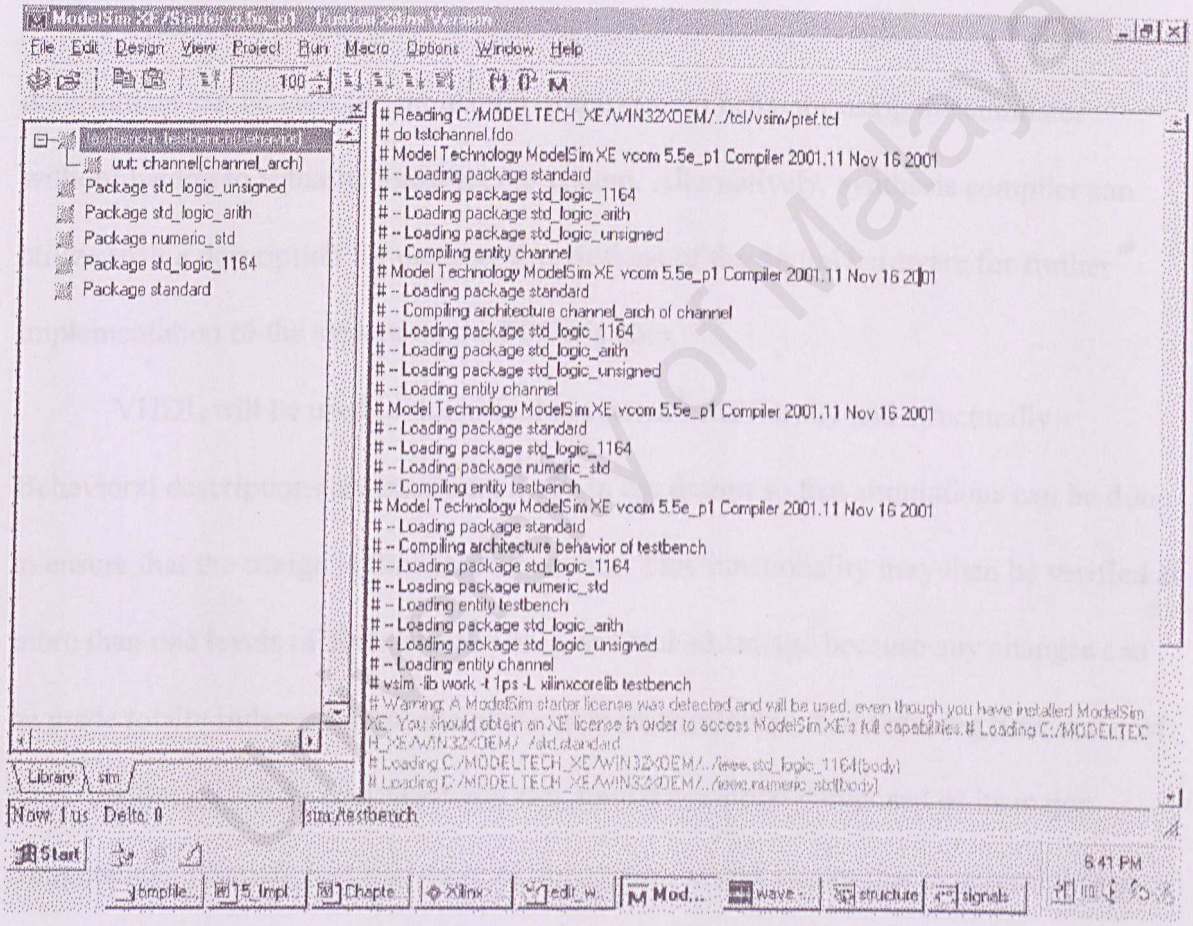


Figure 5.3: An example of the Model Sim XE Transcript Window.

5.3 VHDL and Preprocessing Voice Signal

Previously in Chapter 1, it is mentioned that the scope of the system will determine the range and how the system will work. The main scope of the project is to design a voice recognition system in terms of pre processing the voice signal and then implementing a best voice recognition algorithm and at the same time generating VHDL source code. Individual modules will be defined in terms of its functions and interconnections between them.

VHDL is a language for describing digital systems.[Douglas, 1998] .Such descriptions can be used for simulation of the system behavior using the simulator without having to actually construct the system. Alternatively, synthesis compiler can utilize such a description for creating descriptions of the digital hardware for further implementation of the system onto a FPGA's chip.

VHDL will be used to describe this system behaviorally and structurally. Behavioral descriptions are necessary early in the design so that simulations can be done to ensure that the design is functionally correct. This functionality may then be verified at more than one levels of abstraction. This gives and advantage because any changes can be made totally independent of the physical implementation, thus reducing time and cost. Next, the design can be translated to a structural description composed of its major components. Simulation at this stage of description would ensure that structural design correctly performs the intended functions using the design major components. While there are many hardware description languages prior to VHDL, most of them are developed to serve the simulators that run them. VHDL, on the other hand is technology independent, is notified to a particular simulator and does not enforce a design

methodology on a designer thus making a standard and suitable language, offering benefits over other hardware description languages. Chapter 3 has pointed out the characteristics of VHDL together with its advantages.

5.4 Implementation Steps

The implementation steps were divided into three main stages. First stage is referred to as the 'implementation stage'. It involved in writing the behavioral VHDL description program called the VHDL source code for each portion of the partitioned design in the form of a module. The whole design of the Preprocessing Voice Signal consists of three major modules. Next, is to compile and debug the VHDL codes written. If any errors occur during the compilation, then fixing the errors (or bugs) will be necessary before continuing with the further steps. This step is important in order to ensure the correctness of the VHDL codes.

The second stage is referred to as the 'simulation stage'. A script of 'test bench' will be written for each VHDL codes produces in the implementation stage. The purpose of test bench is to allow the simulation process to take place and establish the clock generator circuit. Following next is the simulation process mentioned to check for functionality of each module available in the design. Lastly, is the functional verification of the modules that includes if there is any module malfunctioning.

The third and last stage is the self-test stage. The circuit will undergo the test process simulation, which put the circuit under testing mode. Throughout the stage, the steps involved are as follows:

- toggling clock cycles generated by test bench

- generating test vectors
- comparing the output signature with a known good signature using test bench.

5.5 Modules Pin Description

Front-end of voice recognition system consists of 3 main modules and 2 small modules:

a) A clock generator module which outputs the left/right channel select signals

b) A channel module which contains the shift registers, buffers, read control, and overflow detection circuitry for a single input stream of data

c) A top-level module, which combines the clock generator module with two channel modules to form a complete codec interface circuit.

d) The clock divider module used for the purpose of slowing the main clock input

e) And the LED Decoder to generate led numbers output

5.5.1 Top level pin description of Preprocessing Voice Signal

Figure 5.4 below describe the input and output pins of the main modules of the Preprocessing Voice Signal system design. Overall there are 5 input pins and 6 output pins incorporated with the design. Each of the input and output pin has its own design specification which will be listed out one by one below.

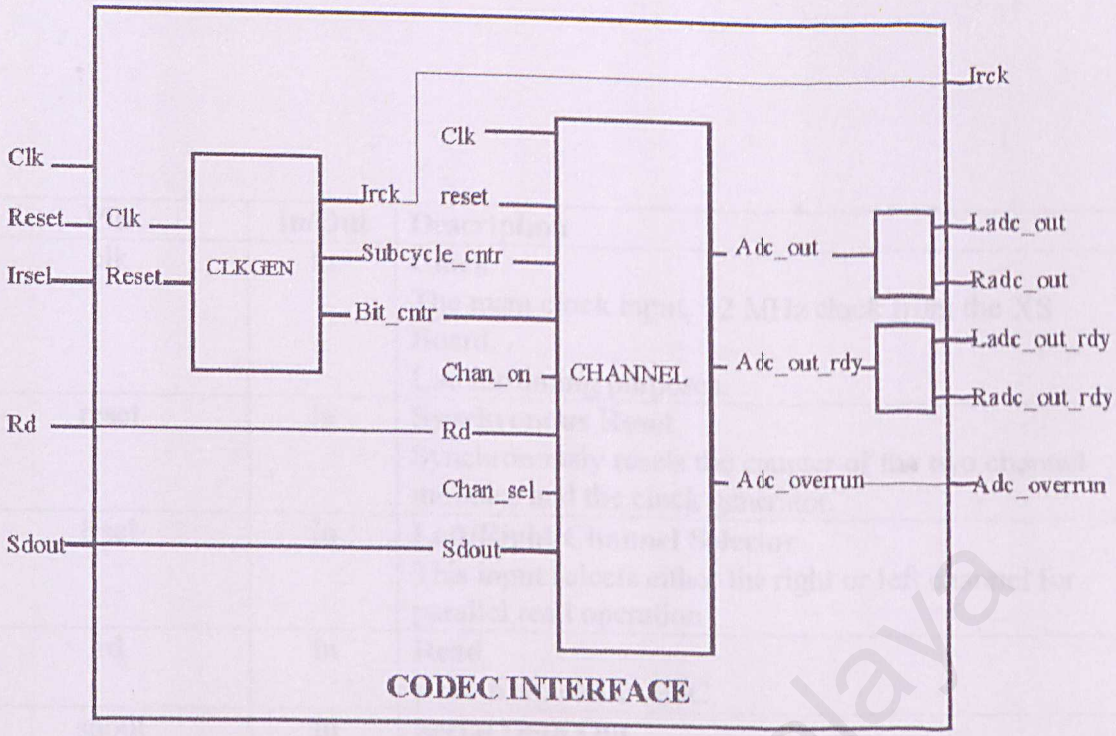
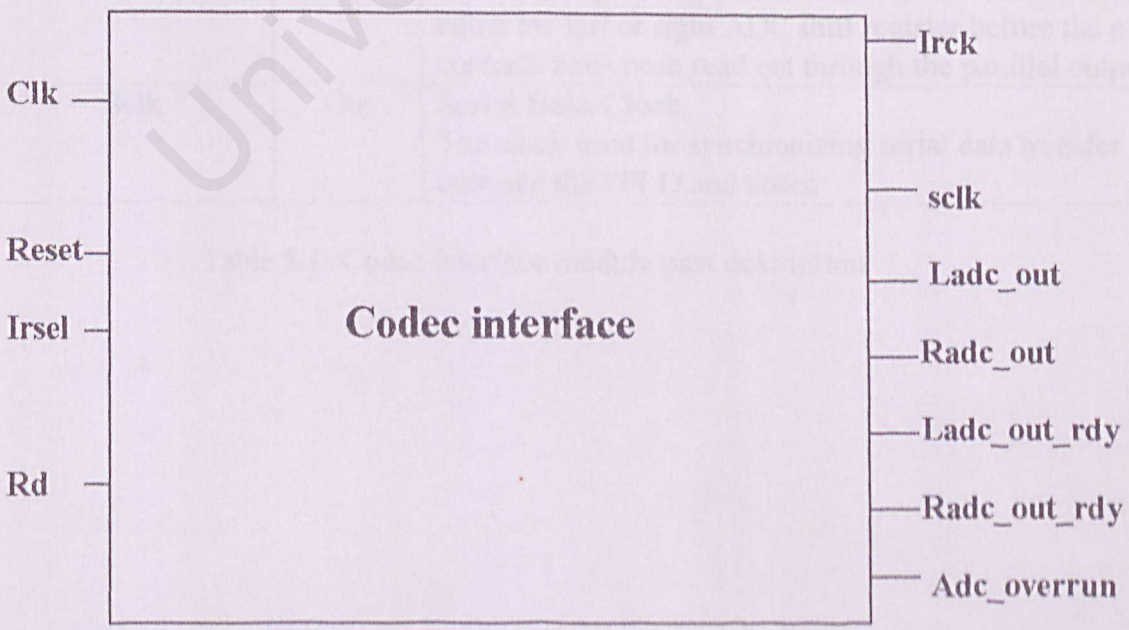


Figure 5.4: Top level pin description of Preprocessing Voice Signal system

5.5.2 Codec interface module Pin Description

It is a top-level module, which combines the clock generator module with two channel modules to form a complete circuit.

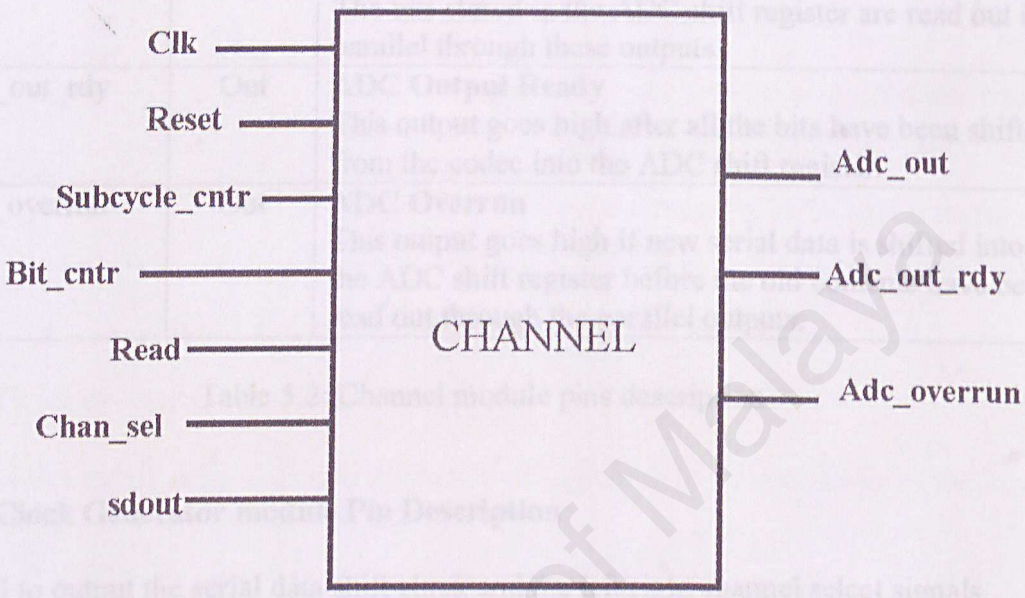


Pin	In/Out	Description
clk	In	**Clock** The main clock input, 12 MHz clock from the XS Board. Use for timing purposes.
reset	In	**Synchronous Reset** Synchronously resets the counter of the two channel modules and the clock generator.
lrsel	In	**Left/Right Channel Selector** This input selects either the right or left channel for parallel read operation
rd	In	**Read** Read from codec ADC
sdout	In	**Serial Data Out** The serial data stream from the codec ADC is shifted in through this input.
ladc_out, radc_out	Out	**Left/Right ADC Output** The bits stored in the left and right ADC shift registers are read out in parallel through these outputs
ladc_out_rdy, rdac_out_rdy	Out	**Left/Right ADC Output Ready** These outputs go high after all the bits have been shifted from the codec into the left or right ADC shift register, respectively.
adc_overrun	Out	**ADC Overrun** This output goes high if new serial data is shifted into either the left or right ADC shift register before the old contents have been read out through the parallel outputs.
Sclk	Out	**Serial Data Clock** The clock used for synchronizing serial data transfer between the FPLD and codec

Table 5.1: Codec interface module pins description

5.5.3 Channel module Pin Description

It contains a shift registers, buffers, read control, and overflow detection circuitry for a single input/output stream of the data.



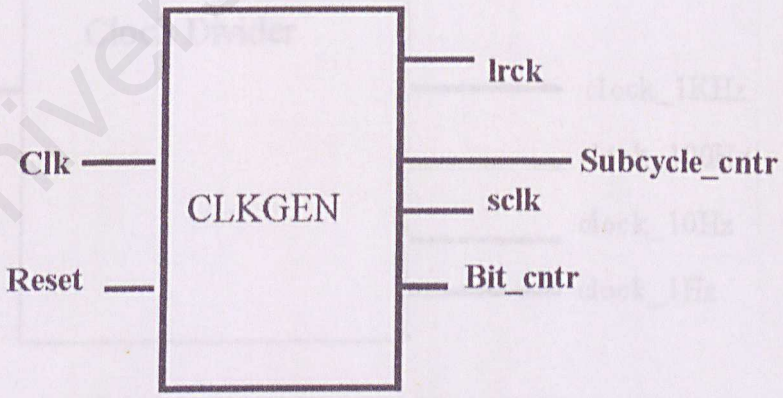
Pin	In/Out	Description
clk	In	**Clock** The main clock input, 12 MHz clock from the XS Board. Use for timing purposes.
reset	In	**Synchronous Reset** Synchronously resets the channel.
chan_on	In	**Channel On** A high level on this input activates the channel. This input is usually controlled by the left/right channel selector.
bit_cntr	In	**Bit Counter** These inputs inform the channel of the index of the serial data bit currently being transmitted and received.
subcycle_cntr	In	**Subcycle Counter** The duration of each serial data bit is divided into four phases and this input indicates the current phase from clock generator module.
chan_sel	In	**Channel Select** A high level on this input enables the interface that lets

		the shift registers be read.
rd	In	**Read** A high level on this input outputs the value stored in the shift register connected to the ADC.
sdout	In	**Serial Data Out** The serial data stream from the codec ADC is shifted in through this input.
adc_out	Out	**ADC Output** The bits stored in the ADC shift register are read out in parallel through these outputs.
adc_out_rdy	Out	**ADC Output Ready** This output goes high after all the bits have been shifted from the codec into the ADC shift register.
adc_overrun	Out	**ADC Overrun** This output goes high if new serial data is shifted into the ADC shift register before the old contents have been read out through the parallel outputs.

Table 5.2: Channel module pins description

5.5.4 Clock Generator module Pin Description

It is used to output the serial data shift clock and the left/right channel select signals.



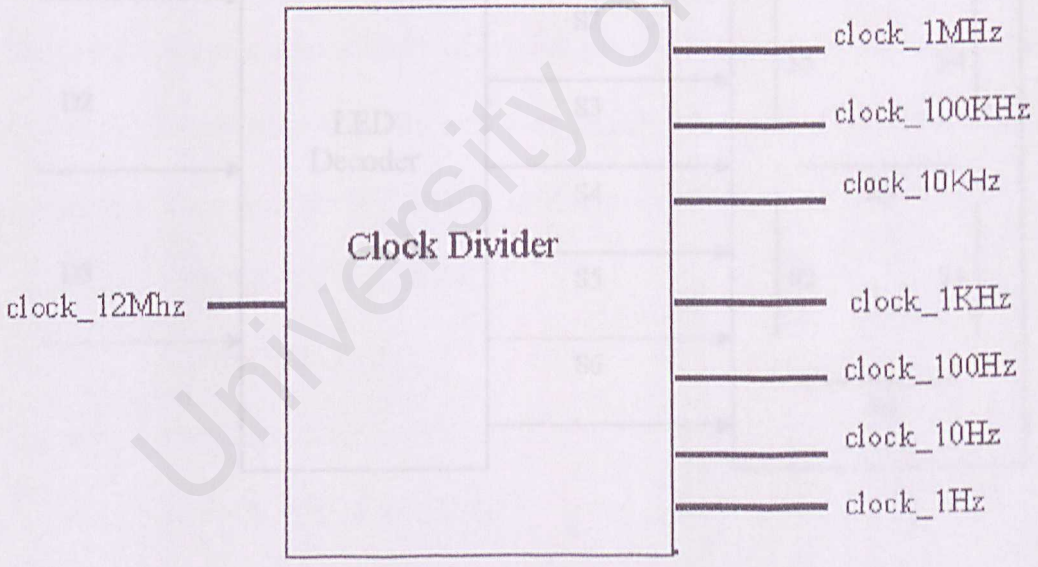
Pin	In/Out	Description
clk	In	**Clock** The main clock input, 12 MHz clock from the XS Board.

		Use for timing purposes.
reset	In	**Synchronous Reset** Synchronously resets the counter of the clock generator.
lrck	Out	**Left/Right Codec Channel Select** This output controls the activation of the left and right channel circuitry in the codec and the FGPA.
bit_cntr	Out	**Bit Counter** These outputs indicate the current bit being transmitted and received in the serial data streams.
subcycle_cntr	Out	**Subcycle Counter** The duration of each serial data bit is divided into four phases and these outputs indicate the current phase
Sclk	Out	**Serial Data Clock** The clock used for synchronizing serial data transfer between the FPLD and codec

Table 5.3: Clock Generator module Pin Description

5.5.5 Clock divider module Pin description

This smaller module is generated to produce a slower clock of 1 Hz that will be used in implementing the voice recognition algorithm.



Pin	In/Out	Description
clock_12Mhz	In	**Clock 12 MHz** The main clock input, 12 MHz clock from the XS Board. Use for timing purposes.
clock_1MHz, clock_100KHz,	Out	**Clock 1 MHz, Clock 100 KHz, Clock 10 KHz, clock 1KHz, Clock 100Hz, Clock 10 Hz and Clock 1 Hz**

clock_10KHz, clock_1KHz, clock_100Hz, clock_10Hz, clock_1Hz		Output the divided clock respectively to give us a slower clock of 1 Hz.

Table 5.4: Clock divider module pins description

5.5.6 LED Decoder Module Pin Description

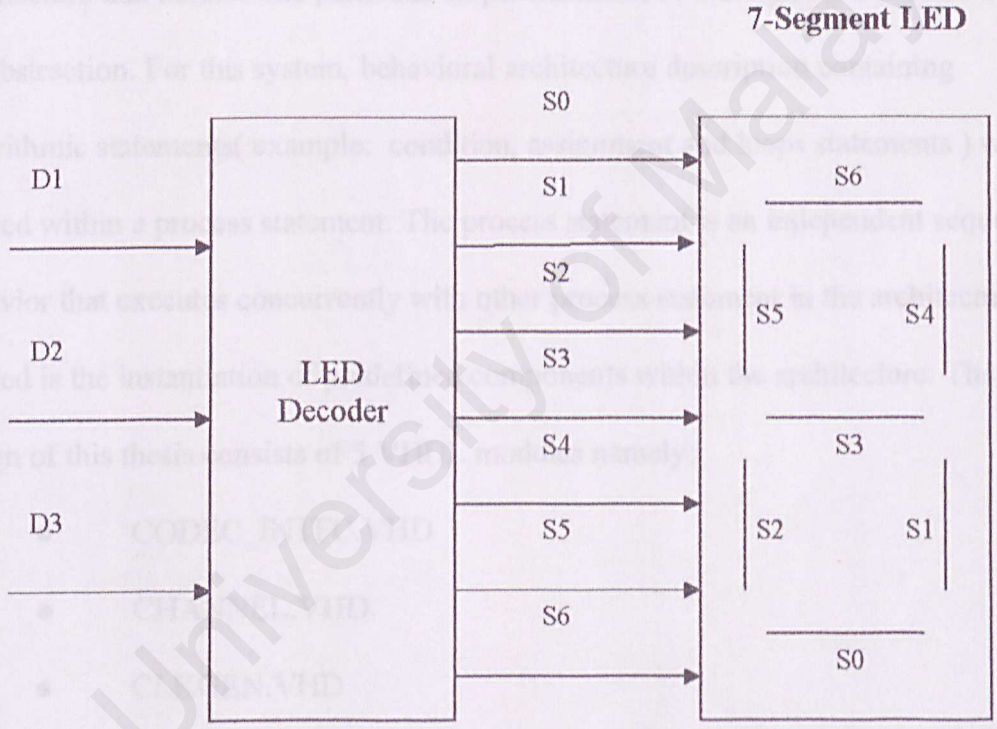


Figure 5.5 : High Level Diagram of the LED Decoder

An LED decoder takes a three bit input from the Decision Module(from Voice Recognition algorithm) and outputs seven signals which drive the segments of an LED

digit. The LED segments will be driven to display the digit corresponding to the value of the three input bits as mentioned in Chapter 4.

5.6 Writing VHDL code

Using VHDL, description of each module in the Preprocessing Voice Signal design starts with an entity statement at the top-level of the VHDL hardware specification hierarchy. The entity statement defines the input and output ports together with the direction of each signal flowing through each port. Associates with entity are the architecture that defines one particular implementation of a design unit, at some level of the abstraction. For this system, behavioral architecture description containing algorithmic statements(example: condition, assignment and loops statements) will be applied within a process statement. The process statement is an independent sequential behavior that executes concurrently with other process statement in the architecture. Also applied is the instantiation of predefined components within the architecture. The overall design of this thesis consists of 5 VHDL modules namely:

- CODEC_INTFC.VHD
- CHANNEL.VHD
- CLKGEN.VHD
- CLOCK_DIVIDER.VHD
- LED.VHD

5.7 Test bench

For the top level entity, a test bench is created as a new module and added in the Xilinx's WebPACK ISE project hierarchy containing all the Preprocessing Voice Signal system's source files. The test bench module instantiates the top level as a unit under test and is used to drive the input pots and subsequently read the output ports of the top level during simulation. Simulation of each module of the design was performed to validate the functional behavior of each portion as well as for the top level. Normal and testing operation will be done to validate functional behavior during system testing phase. The simulation process and the related results will be discussed in detail in the next chapter.

```
BEGIN
uut: channel PORT MAP(
                    clk => clk,
                    reset => reset,
                    chan_on => chan_on,
                    bit_cntr => bit_cntr,
                    subcycle_cntr => subcycle_cntr,
                    chan_sel => chan_sel,
                    rd => rd,
                    adc_out => adc_out,
                    adc_out_rdy => adc_out_rdy,
                    adc_overrun => adc_overrun,
                    sdout => sdout
          );
  CLOCK: process
  begin
    Clock_cycle <= Clock_cycle + 1;
    Clk <= '1';
    wait for 41.5 ns;
    Clk <= '0';
    wait for 41.5 ns;
  end process;
-- *** Test Bench - User Defined Section ***
  tb : PROCESS
  BEGIN

--bit_cntr =0
                              reset <= '1';
                              chan_on<='1';
          bit_cntr<="000000";
                    subcycle_cntr<="10";
                    chan_sel<='0';
```

Figure 5.6: A portion of Channel.vhd test bench module

Chapter 6 : System Testing

6.1 Introduction

So far discussions covered presentation to the reader on how the Preprocessing Voice Signal has been implemented. Continuing in this chapter is to present reader with the systems testing or system validation in order to make sure the system is functioning as expected theoretically. System testing involves simulation process towards all the modules modeled in VHDL descriptions. This chapter basically discusses on all the steps considered during system testing beginning with simulating smaller and end at the simulation process of high-level module integrated with smaller modules.

6.2 Design Simulation

Design Simulation starts with checking for any syntax error in VHDL codes written. If there is any syntax errors, modifications must be made to the source codes and the process take place repeatedly. The next step is to link all the source files followed by simulating the files linked together. Simulation process is carried out using Model Sim Simulator. Simulation results will be generated in the form of output waveforms of the VHDL simulation run and displayed in the form of output waveforms of the VHDL simulation run and displayed in the Model Sim 'Waveform' display window for system validation.

6.3 Clock Generator module Simulation

Clock Generator module is use to output the left/right channel select signals. There are 2 main clocks that will be used throughout the system, which are the Lrck and

Sclk (Serial Interface Clock). In the test bench we start the simulation with the left channel being selected first. (**Irck** input port is set to 'left')

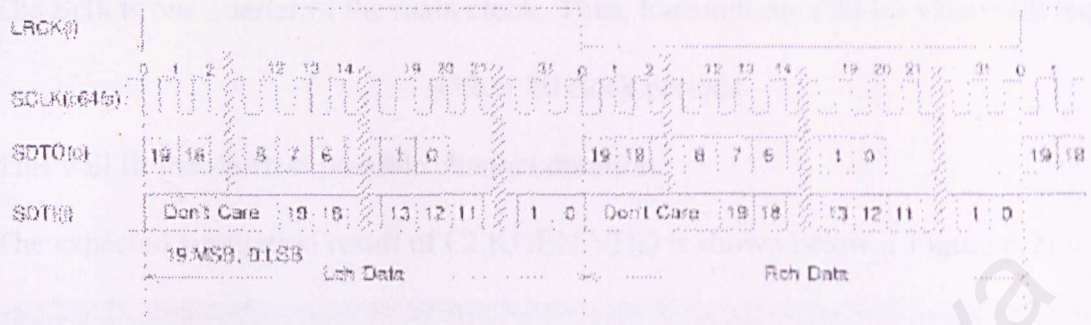


Figure 6.1: Timing diagram of the main clock used in CLKGEN.VHD

In the module, there is a process to increment the sequencing counter and toggles the left/right channel selector when the count reaches the duration for which a channel is active. The codec chip requires that the channel duration be 128 clock periods in length. Therefore, to handle left channel alone requires 128 clock periods, and so does the right channel. This means that the left channel will be active for 128 clock periods. After 128 clock periods, the channel will be toggled, thus the **right** port will then be high. Thus, the total time to handle both channels is 256 clock periods and each port will take turn to be high for 128 clock periods. (Figure 6.1)

The **Bit Counter** output port that is used in this module is a 6-bit port. To complete a single channel duration of 128 clock periods, it will output the position of the current data bit in the serial stream in the length of 32-phase, where each phase consists of 4-subcyle counter. Thus, to handle both channels, a 6-bit port of bit counter is used, where

2^6= 64 (32 phases for each channel duration)

The **Subcycle Counter** output port is used to output the position within a bit. It will be given by the two LSB of the sequence counter.

The **Sclk** is one quarter of the main clock. Thus, transmitting a 20-bit value will require

4*20= 80 clock periods

This will fit the shortest possible channel duration.

The expected simulation result of CLKGEN.VHD is shown below. (Figure 6.2)

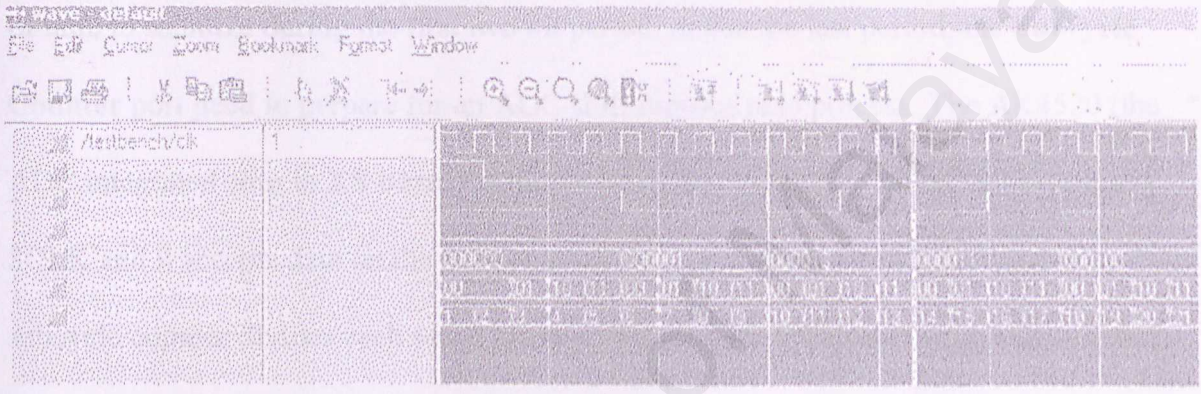


Figure 6.2: Simulation result of CLKGEN.VHD module

6.4 Channel module Simulation

There are 3 main processes in this module:

- Receives data from codec ADC
- Handle reading of ADC data from codec interface
- Detect overrun process

6.4.1 Receives data from codec ADC

This process is to receive serial data from the ADC in the codec. The ADC shift register is cleared upon reset and a flag is set which indicates the shift register does not

contain all the bits from the ADC. Once the reset is removed and the channel is on/active (**Channel** port= '1' **Sdout** port = '1'), bits are shifted into the register.

Bits 1,2,..., up to the width of the ADC, (ADC width = 20) data value are pushed into the shift register. Then the shifting stops. Bits are shifted into the register during the third subcycle of each bit period (the subcycles are numbered 0, 1, 2 and 3 where each one is equivalent to "00", "01", "10" and "11"). This is because serial data bit will have a plenty of time to stabilize during the first two bit period, and at the last period, the **Subcycle Counter** port need to prepare for an ADC shift register read process. The AK4520 (the ADC integrated chip in XS board) outputs data on the SDTO pin on the falling edge of SCLK and it accepts data on the SDTI pin on the rising edge of SCLK. (Figure 6.1) The subcycle counter divides each serial bit into four phases. When subcycle_cntr = 2, then this is halfway between a falling edge of SCLK and a rising edge of SCLK. This is an appropriate instant for the FPGA to get any data output by the AK4520 on SDTO and send new data to the AK4520 through SDTI. There will be no outputting and inputting serial data at the right time if the trigger value of the subcycle_cntr is change.

The shift register is marked as 'not full' as soon as a single bit is shifted in so that the value will not be inadvertently read. The shift register status changes to full as soon as the last bit enters the shift register.

For the first test bench (TSTCHANNEL_RCVADC.VHD), testing will be done based on 01, 2,.....up to the 18^{th} bit value to be push to the shift register. Then the contents of

the shift register will be output at the **adc_out** port in a parallel format. These outputs are not latched and will change as bits are shifted into the register. (Figure 6.3)

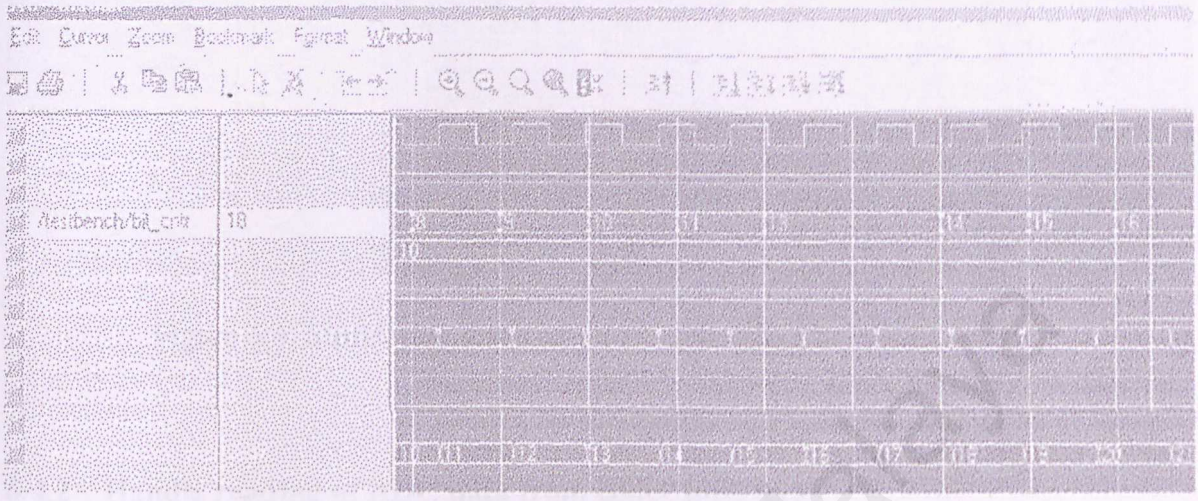


Figure 6.3: Simulation result of TSTCHANNEL_RCVADC.VHD

For the second test bench, (TSTCHANNEL_RCVADC1.VHD) the different value of every input port will be set in order to test the functionality of the Channel module. From the simulation result (Figure 6.4), it shows that the **chan_on** and **sdout** port must be set to 'high' value in order to make sure the bit shifting process is being done. As mentioned in Clock Generator module, the value of **Subcycle_cntr** must also set to the value of 2 ('10' in bit) to make sure that there will be an inputting of serial data. Meanwhile, the **chan_sel** and **rd** input port has no effect in the receiving data process. This is because both of them are only involved in handling the reading process as will be describe in the next section.

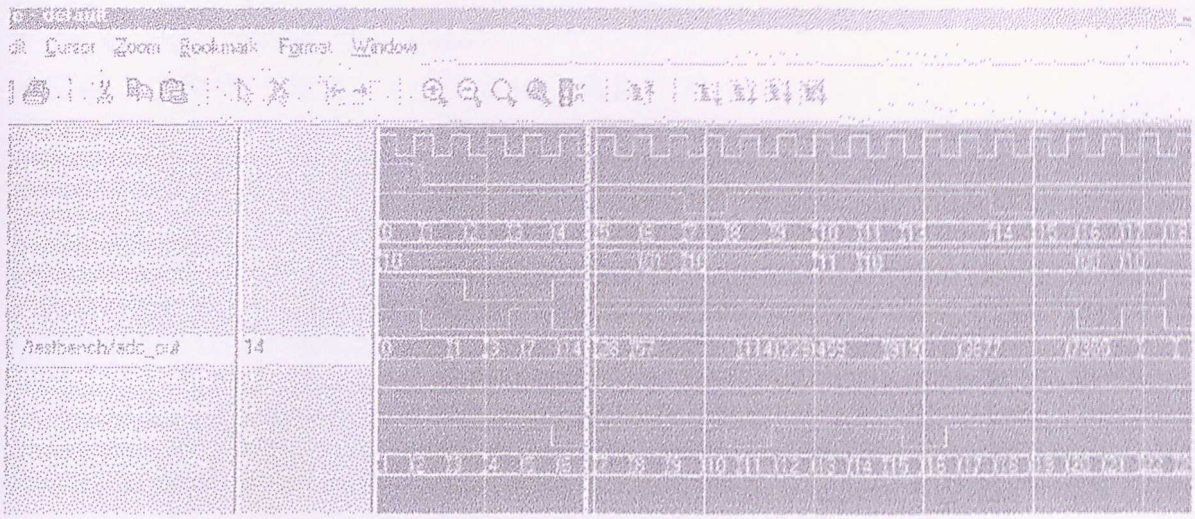


Figure 6.4: Simulation result of TSTCHANNEL_RCVADC1.VHD

6.4.2 Handle reading of ADC data from codec interface

As describe in Chapter 4, a flag will then be maintained to indicate whether the contents of the ADC shift register have been read. The flag is set when the ADC register for the channel is full and it is selected for a read operation. The flag will stay set after the read operation is complete. Reading the register does not empty it. The shift register is no longer full only when the first bit of the next sample is shifted into it. This will reset the read flag.

The **read_adc process** will updates the flag that indicates whether the ADC shift register has been read. A status output is asserted when the data in the ADC shift register is ready for reading. Reads are permitted when the register is full and has not yet been read. Register is full when it reaches the width of ADC. Thus, when the reading process is executed, the **adc_out_ready** output port will be on 'high' value. This is shown in the first test bench TSTCHANNEL_READOUT.VHD. (Figure 6.5)

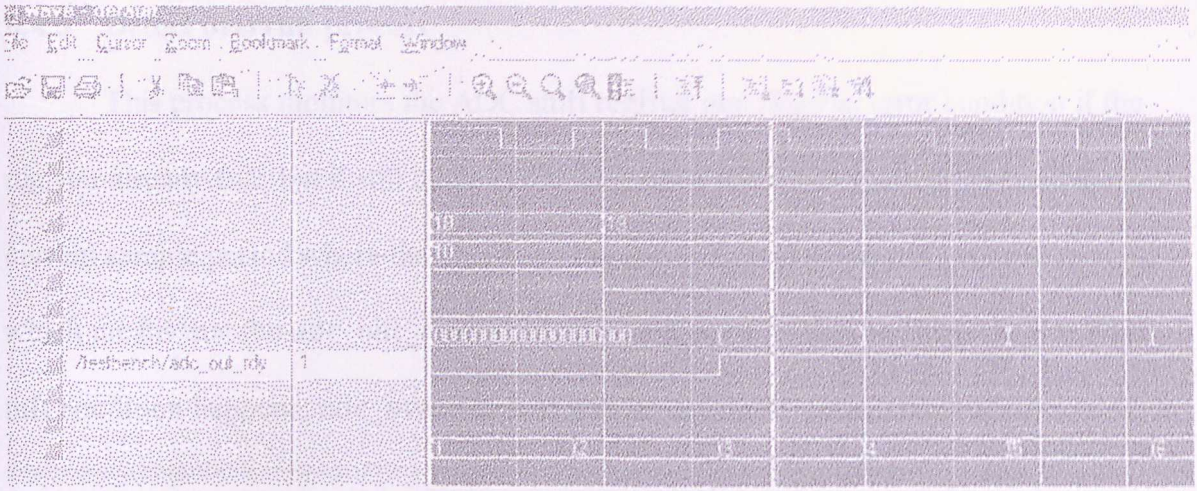


Figure 6.5: Simulation result of TSTCHANNEL_READOUT.VHD

The second test bench will show that the **adc_out_ready** output is cleared as soon as a read occurs or new data is shifted into the register. This happen when the **chan_sel** port= '1' AND **Read** port= '1'. Besides that, the result also shows that other input ports have no effect on the **adc_out_ready** output. (Figure 6.6)

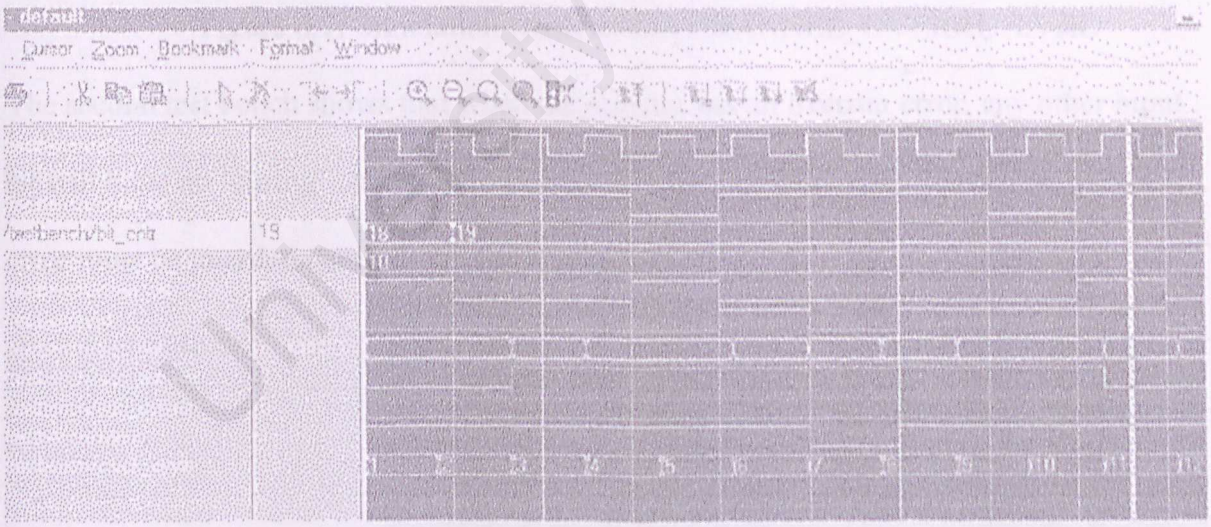


Figure 6.6: Simulation result of TSTCHANNEL_READOUT1.VHD

6.4.3 Detect overrun process

This process monitors the ADC shift register and flags an error condition if the register already full but it still begins accepting bits (when bit counter =1) from the current sample period where the data from the previous period has not yet been read, which means that the **adc_out_ready** output port is must be still on 'high' value. When this condition happens, **adc_overrun** port will be on 'high' value. (Figure 6.7)

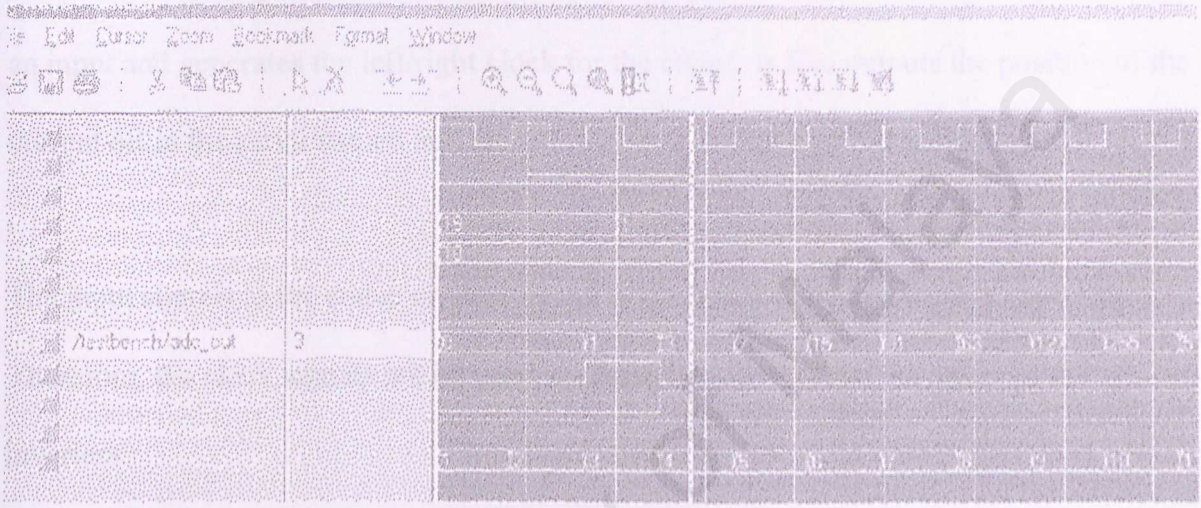


Figure 6.7: Simulation result of TSTCHANNEL_OVERRUN.VHD

The second test bench shows that when the register has an overrun error, any other input bit will have no 'curing' condition to correct the error. (Figure 6.8). Therefore, overrun error must be guarded carefully by the system user.

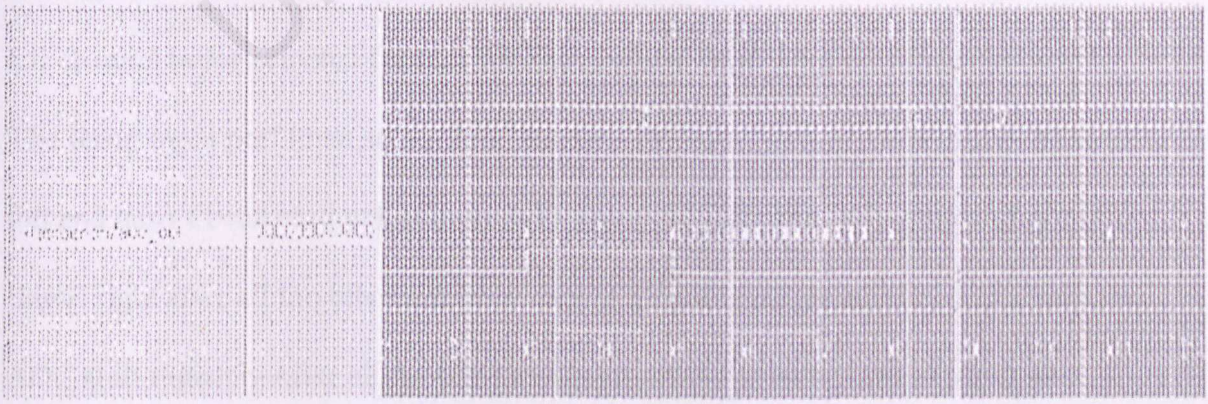


Figure 6.8: Simulation result of TSTCHANNEL_OVERRUN1.VHD

6.5 Codec Interface module Simulation

Codec Interface is a top-level module, which combines the clock generator module with two channel modules to form a complete circuit. The 3 main processes that occur in this module are as follows:

6.5.1 u0 process

One clock generator module will be instantiated. It receives the 12 MHz clock as an input and generates the left/right clock for the codec. It also outputs the position of the current bit in the serial stream and the current cycle within each bit period.

The input signals to the codec on the XStend V1.3 Board will pass through inverters. Therefore, the clock signals are inverted on these lines to remove the effect of the inverters.

6.5.2 u_left and u_right process

This module, which handles the left channel of the codec, will be instantiated. This module is activated during one half of the left/right clock period. It is selected for reading by the left/right selection input. The test bench starts the simulation test with the left channel being selected first for read operation (**lrsel** input port is set to 'low'). The **Read** input port must set to 'low' because the ADC data is ready when the register is full and hasn't been read yet. (Figure 6.9) Thus, the **ladc_out_rdy** output port will be 'high' right after all the bits have been shifted from the left channel of codec into the ADC shift register. After all the bits have been shifted from the left channel, the **radc_out** will then output the contents of the right channel.

The **lrck** output port will then be 'high' for 128 clock periods representing the right clock signals being activated, effect from the inverters. After the first 128 clock periods, the **lrck** will be 'low' for another 128 clock periods. This goes on until the whole process of simulating ends.

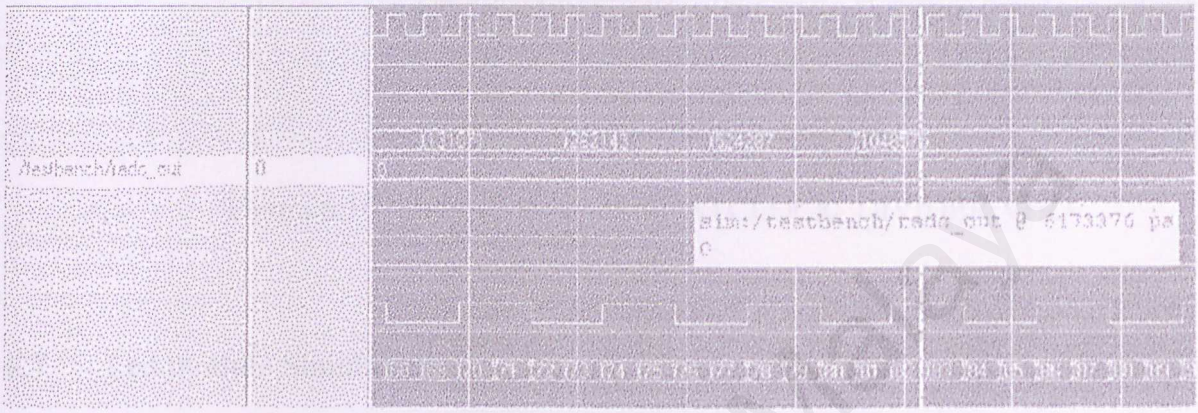


Figure 6.9: Simulation result of TSTCODEC_INTFC.VHD

6.5.3 Overrun error

The overrun error indicator for the codec interface is formed by the logical-OR of the associated error outputs of the left and right channel modules. Thus an error is reported if either channel reports an error.

6.6 Clock Divider module Simulation

Using test bench, a slower clock running at 1Hz can be produce from the clock divider module. If we use one huge synchronous parallel counter, all the bits will change simultaneously, causing a huge power pulse that will cause the chip to malfunction. The solution is to use a small synchronous counter with a few bits to divide the clock down to about 1MHz, and then use a slower ripple counter to divide that 1MHz clock down to the

1Hz. The ripple counter allows each bit to change at different times, eliminating the big power pulse and allowing the chip to function correctly. The simulation result can be seen from Figure 6.10.

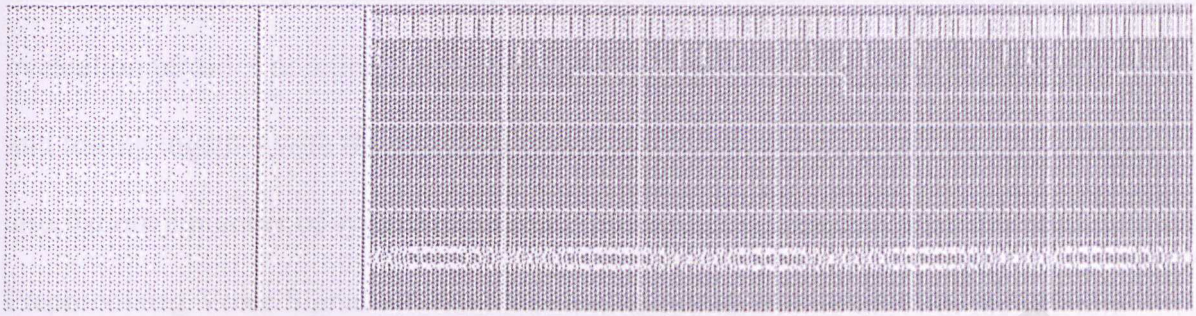


Figure 6.10: Simulation result of TSTCLOCK_DIVIDER.VHD

6.7 Led module Simulation

An LED decoder takes a three bit input from the Decision Module (from Voice Recognition algorithm) and outputs seven signals which drive the segments of an LED digit. In the test bench, there will be only 3 value tested which are 0,1 and 2. The remaining will be kept for reserve used. (Figure 6.11)

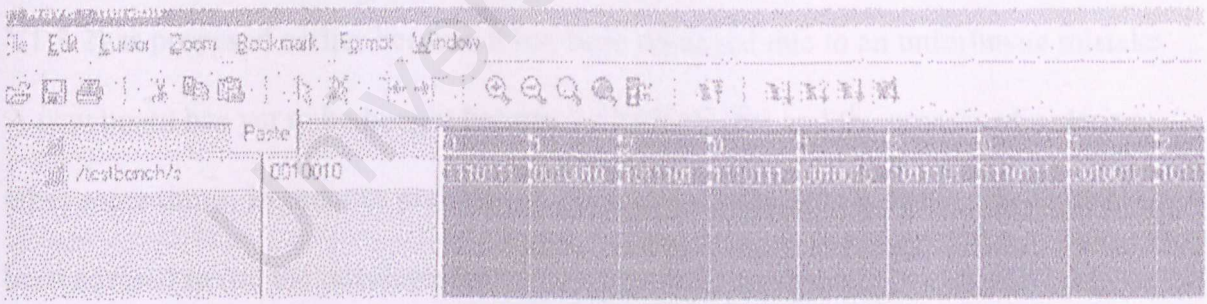


Figure 6.11: Simulation result of TSTLED.VHD

Chapter 7: System Evaluation

7.1 Introduction

This chapter presents the Preprocessing Voice Signal systems evaluation. System evaluation includes pointing out the problems encountered during system implementation and testing together with the solutions considered in solving the problems. This chapter also list out a number of system strengths as well as the constraints related to the Preprocessing Voice Signal system. Ideas and theory on how to improve the existing system design are also included as for future enhancements considering the recent technology could be applied to the system in future. Lastly, it will also cover on the knowledge and experience gained throughout developing this system.

7.2 Problems Encountered and Solutions

One of the main problems faced is that the Preprocessing Voice Signal system design could not be implemented into the chosen development tools (Xstend Board V1.3.2) as proposed earlier because it has been damaged due to an unfortunate mistake. A new board has yet to be bought because of its high cost and the process of ordering takes some times. In order to prevent this issue from happening in future, students has been advised to use the development tools with extra carefulness.

System implementation marks the lack of mastery in both modeling Preprocessing Voice Signal system using VHDL description and the Xilinx's WebPACK ISE software. Lack of mastery in VHDL programming language and the new environment of the Model Sim XE simulation tools create errors during compilation of modules in the Preprocessing Voice Signal system. The system cannot be link result from these

compilation errors. This causes a suspending simulation. Simulation strategy in which is to compile the modules from the lowest level subsystem helps to reduce much more complex errors that may be produced during higher level subsystem's of VHDL description. Besides that, there is only one operation of each module can be tested at one time when using Xilinx's WebPACK ISE compiler. This approach risks in the time taken to build each successful subsystem module. The more time spent on each of the subsystem module, the longer it takes to complete the whole Preprocessing Voice Signal system.

Another problem is that, each successfully compiled module does not mean that the module functions correctly as expected. It is necessary to look for logical errors when there is a faulty value shown in the output waveform generated by the compiler.

7.3 System Strengths

Preprocessing Voice Signal system highlights the use of Codec ADC in XStend Board. It is highly integrated with high performance to provide stereo analog-to-digital converters using delta-sigma conversion technique. Applications include reverb processors, musical instruments, DAT and multi rack recorders. Thus, it is a very suitable device to be used in implementing the Preprocessing Voice Signal system.

This system also provides an easy use of voice recognition in preprocessing voice signal by using VHDL as the hardware programming language. The coding is simple and easy to understand even to a first time user. By using VHDL, this design can be used documentation, verification, and synthesis of larger digital designs. VHDL is also

designed to be device-independent with behavioral simulation that permits design optimization.

7.4 System Constraints

One of the Preprocessing Voice Signal system drawbacks is the overrun error that may occur. As already mentioned in Chapter 4, an overrun error occurs when new data arrives when the receive buffer is full (there is no place to put the new data). The data that caused the overrun condition to be detected is lost and the last good data character that was received is flagged with the overrun error. It indicates whether new data sent in is overwriting the previous data received that has not been read out yet. The potential for data overruns, however, is always present. Data overruns must be guarded against since the overrun error condition can only be detected after one or more data characters have already been lost. The *parity error*, *framing error*, and *overrun* error indicate any problems with the current received data.

7.5 Future Enhancements

This section focuses on improvements that can be made to the existing Preprocessing Voice Signal system. The main enhancements that can will done is to program the chosen development board with the produced behavioral coding so as to provide a clearly view of how the system actually works. This can be done as soon as the new Xstend Board V1.3.2 is bought.

Other than that, focus has been put on the displaying part, where 'vocabulary' and 'sentences' will be implemented in voice recognition to provide wide usage of Preprocessing Voice Signal system, instead only on numbers.

Another improvement that can be made is to make sure the Preprocessing Voice Signal system may receive input from noisy environment as well, by implementing filters.

7.6 Knowledge and Experience Gained

During the four consecutives months of developing this system, exposure of what it needs to develop a Preprocessing Voice Signal system from scratch has been get. The related terms to the topic given has been searched after getting the topic on the first day. Basic concepts on terms such as Analog-to-Digital converter, Serial and Parallel data, Shift Register, Clock divider and many more were studied to provide a better understanding in developing this system. Discussion sessions with Supervisor were attended as frequent as possible to discuss issues related to the topic. A design proposal was produced and presented to the Moderator. This continues with the design implementation using VHDL programming language and Xilinx's WebPACK ISE simulation tools. This continues with the testing of the system for its functionality and verification. Again, a successful system was presented to the Moderator. The presentation marks the last part of the system development process. It is noticed that during this period, self-discipline and teamwork are very important to make sure the system is successful.

Conclusion

Preprocessing Voice Signal system has an advantage of using VHDL as the programming language because of its portability, standardized, productivity, reusability and lower cost of implementation benefits. Throughout this thesis, the first four chapters have been presented on the design issues and review on existing designs. Chapter 5 explains in detail of how the system is being implemented with VHDL behavioral and structurally description. Chapter 6 presents the simulation resulted from the Preprocessing Voice Signal system obtained in the form of waveform output. Positive results shown in the waveform output verifies that the system is successful. Having simulated and functionally verifies, it has been proven in this thesis that the Preprocessing Voice Signal system is a successful system.

Appendices

System Coding

Description/ File name: CLKGEN.VHD

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;


ENTITY clkgen IS
GENERIC
(
CHANNEL_DURATION: positive := 128 -- must be 128
);
PORT
(
-- interface I/O signals
clk: IN std_logic; -- clock input
reset: IN std_logic; -- synchronous active-high reset
-- codec chip clock signals
sclk: OUT std_logic; --serial data clock to codec
lrck: OUT std_logic; -- left/right codec channel select
bit_cntr: OUT std_logic_vector(3 DOWNTO 0);
subcycle_cntr: OUT std_logic_vector(1 DOWNTO 0)
);
END clkgen;

ARCHITECTURE clkgen_arch OF clkgen IS
CONSTANT yes: STD_LOGIC := '1';
CONSTANT no: STD_LOGIC := '0';
CONSTANT ready: STD_LOGIC := '1';
CONSTANT overrun: STD_LOGIC := '1';
CONSTANT left: STD_LOGIC := '0';
CONSTANT right: STD_LOGIC := '1';
SIGNAL lrck_int: std_logic;
SIGNAL seq: std_logic_vector(7 DOWNTO 0);
BEGIN
gen_clock:
PROCESS(clk,seq,lrck_int)
BEGIN
IF (clk'event AND clk='1') THEN
IF(reset=YES) THEN -- synchronous reset
seq <= (OTHERS=>'0');
lrck_int <= left; -- start with left channel of codec
ELSIF(seq=CHANNEL_DURATION-1) THEN
seq <= (OTHERS=>'0'); -- reset sequencer every channel period
lrck_int <= NOT(lrck_int); -- toggle channel sel every period
ELSE
seq <= seq+1;
lrck_int <= lrck_int;
END IF;
END IF;
END PROCESS;
lrck <= lrck_int; -- output the channel selector to the codec
sclk<=seq(1); -- serial data shift clock equals input clock
bit_cntr <= seq(7 DOWNTO 2);
subcycle_cntr <= seq(1 DOWNTO 0);
END clkgen_arch;
```

Description/ File name: TSTCLKGEN.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

        COMPONENT clkgen
        PORT(
clk: IN std_logic; -- clock input
reset: IN std_logic; -- synchronous active-high reset
sclk: OUT std_logic;
lrck: OUT std_logic; -- left/right codec channel select
bit_cntr: OUT std_logic_vector(5 DOWNTO 0);
subcycle_cntr: OUT std_logic_vector(1 DOWNTO 0)
                );
        END COMPONENT;

        SIGNAL clk :  std_logic;
        SIGNAL reset :  std_logic;
        SIGNAL sclk:  std_logic;
        SIGNAL lrck:  std_logic;
        SIGNAL bit_cntr :  std_logic_vector(5 downto 0);
        SIGNAL subcycle_cntr :  std_logic_vector(1 downto 0);
        Signal Clock_cycle: natural := 0;
BEGIN

        uut: clkgen PORT MAP(
                clk => clk,
                reset => reset,
                sclk=>sclk,
                lrck=>lrck,
                bit_cntr => bit_cntr,
                subcycle_cntr => subcycle_cntr
        );

  CLOCK: process
  begin
    Clock_cycle <= Clock_cycle + 1;
    Clk <= '1';
    wait for 41.5 ns;
    Clk <= '0';
    wait for 41.5 ns;
  end process;

-- *** Test Bench - User Defined Section ***
  tb : PROCESS
  BEGIN
                reset <= '1';
                wait for 100 ns;
                reset <= '0';
                wait;

END PROCESS;

-- *** End Test Bench - User Defined Section ***
END;
```

Description/ File name: CHANNEL.VHD

```
LIBRARY IEEE;
```

```
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;


ENTITY channel IS
GENERIC
(

ADC_WIDTH: positive := 20
);
PORT
(
-- interface I/O signals
clk: IN std_logic; -- clock input
reset: IN std_logic; -- synchronous active-high reset
chan_on: IN std_logic;
bit_cntr: IN std_logic_vector(5 DOWNTO 0);
subcycle_cntr: IN std_logic_vector(1 DOWNTO 0);
chan_sel: IN std_logic; -- select L/R channel for read/write
rd: IN std_logic; -- read from the codec ADC
sdout: IN std_logic; -- serial input from codec ADC
adc_out: OUT std_logic_vector(ADC_WIDTH-1 DOWNTO 0); -- ADC output
adc_out_rdy: OUT std_logic; -- ADC output is ready to be read
adc_overrun: OUT std_logic -- ADC overwritten before being read
);
END channel;

ARCHITECTURE channel_arch OF channel IS

CONSTANT yes: STD_LOGIC := '1';
CONSTANT no: STD_LOGIC := '0';
CONSTANT ready: STD_LOGIC := '1';
CONSTANT overrun: STD_LOGIC := '1';

CONSTANT left: STD_LOGIC := '0';
CONSTANT right: STD_LOGIC := '1';

SIGNAL adc_shfreg: std_logic_vector(ADC_WIDTH-1 DOWNTO 0);
SIGNAL adc_full: std_logic; -- ADC shift register is full
SIGNAL adc_rd: std_logic; -- the ADC channel has been read
SIGNAL adc_rd_nxt: std_logic; -- the ADC channel has been read
SIGNAL adc_out_rdy_int: std_logic; -- internal version adc_out_rdy
BEGIN
-- receives data from codec ADC
rcv_adc:
PROCESS(clk,chan_on,subcycle_cntr,bit_cntr,adc_shfreg,sdout)
BEGIN
IF(clk'event AND (clk=YES)) THEN
IF(reset='1') THEN
adc_shfreg <= (OTHERS=>'0');
adc_full <= NO;
ELSIF((chan_on=YES) AND (subcycle_cntr=2)) THEN
IF(bit_cntr<ADC_WIDTH-1) THEN
adc_full <= NO;
adc_shfreg <= adc_shfreg(ADC_WIDTH-2 DOWNTO 0) & sdout;
ELSIF(bit_cntr=ADC_WIDTH-1) THEN
adc_full <= YES;
adc_shfreg <= adc_shfreg(ADC_WIDTH-2 DOWNTO 0) & sdout;
END IF;
END IF;
END IF;
END PROCESS;
adc_out <= adc_shfreg;

-- handle reading of ADC data from codec interface
adc_rd_nxt <= YES WHEN (adc_full=YES AND chan_sel=YES AND rd=YES) OR
(adc_full=YES AND adc_rd=YES)
ELSE NO;
read_adc:
PROCESS(clk,adc_rd_nxt)
```

```
BEGIN
IF(clk'event AND clk='1') THEN
IF(reset=YES) THEN
adc_rd <= NO;
ELSE
adc_rd <= adc_rd_nxt;
END IF;
END IF;
END PROCESS;
-- ADC data is ready if register is full and hasn't been read yet
adc_out_rdy_int <= YES WHEN adc_full=YES AND adc_rd=NO ELSE NO;
adc_out_rdy <= adc_out_rdy_int;

-- detect and signal overwriting of data from the codec ADC channels
detect_adc_overrun:
PROCESS(clk,reset,bit_cntr,chan_on,adc_out_rdy_int)
BEGIN
IF(clk'event AND clk='1') THEN
IF(reset=YES) THEN
adc_overrun <= NO;
ELSIF(bit_cntr=1 AND chan_on=YES AND adc_out_rdy_int=YES) THEN
adc_overrun <= YES;
END IF;
END IF;
END PROCESS;
END channel_arch;
```

Description/ File name: TSTCHANNEL_RCVADC.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

        COMPONENT channel
        PORT(
                clk : IN std_logic;
                reset : IN std_logic;
                chan_on : IN std_logic;
                bit_cntr : IN std_logic_vector(5 downto 0);
                subcycle_cntr : IN std_logic_vector(1 downto 0);
                chan_sel : IN std_logic;
                rd : IN std_logic;
                sdout : IN std_logic;
                adc_out : OUT std_logic_vector(19 downto 0);
                adc_out_rdy : OUT std_logic;
                adc_overrun : OUT std_logic
                );
        END COMPONENT;

        SIGNAL clk :  std_logic;
        SIGNAL reset :  std_logic;
        SIGNAL chan_on :  std_logic;
        SIGNAL bit_cntr :  std_logic_vector(5 downto 0);
        SIGNAL subcycle_cntr :  std_logic_vector(1 downto 0);
        SIGNAL chan_sel :  std_logic;
        SIGNAL rd :  std_logic;
        SIGNAL adc_out :  std_logic_vector(19 downto 0);
        SIGNAL adc_out_rdy :  std_logic;
        SIGNAL adc_overrun :  std_logic;
        SIGNAL sdout :  std_logic;
 SIGNAL Clock_cycle: natural := 0;

BEGIN
```

```
        uut: channel PORT MAP(
                clk => clk,
                reset => reset,
                chan_on => chan_on,
                bit_cntr => bit_cntr,
                subcycle_cntr => subcycle_cntr,
                chan_sel => chan_sel,
                rd => rd,
                adc_out => adc_out,
                adc_out_rdy => adc_out_rdy,
                adc_overrun => adc_overrun,
                sdout => sdout
        );

   CLOCK: process
   begin
     Clock_cycle <= Clock_cycle + 1;
     Clk <= '1';
     wait for 41.5 ns;
     Clk <= '0';
     wait for 41.5 ns;
   end process;

-- *** Test Bench - User Defined Section ***

tb : PROCESS   --: rcv_adc ( receive data from ADC)

BEGIN

--bit_cntr =0
                reset <= '1';
                chan_on<='1';
                bit_cntr<="000000";
                subcycle_cntr<="10";
                chan_sel<='1';
                rd<='1';
                sdout<='1';
               wait for 100 ns;

--bit_cntr = 1
                reset <= '0';
                chan_on<='1';
                bit_cntr<="000001";
                subcycle_cntr<="10";
                chan_sel<='1';
                rd<='0';
                sdout<='1';
               wait for 100 ns;

--bit_cntr = 2
                reset <= '0';
                chan_on<='1';
                bit_cntr<="000010";
                subcycle_cntr<="10";
                chan_sel<='1';
                rd<='0';
                sdout<='1';
                wait for 100 ns;


--bit_cntr = 3
                chan_on<='1';
                 bit_cntr<="000011";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='1';
                sdout<='1';
               wait for 100 ns;

--bit_cntr =4
```

```
                chan_on<='1';
                bit_cntr<="000100";
                subcycle_cntr<="10";
                chan_sel<='1';
                rd<='0';
                sdout<='1';
                wait for 100 ns;

--bit_cntr =5
                chan_on<='1';
                bit_cntr<="000101";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='1';
                sdout<='1';
                wait for 100 ns;

--bit_cntr =6
                chan_on<='1';
                bit_cntr<="000110";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='1';
                sdout<='1';
                wait for 100 ns;

--bit_cntr =7
                chan_on<='1';
                bit_cntr<="000111";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='1';
                sdout<='1';
                wait for 100 ns;
--bit_cntr =8
                chan_on<='1';
                bit_cntr<="001000";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='1';
                sdout<='1';
                wait for 100 ns;

--bit_cntr =9
                chan_on<='1';
                bit_cntr<="001001";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='1';
                sdout<='1';
                wait for 100 ns;

--bit_cntr =10
                chan_on<='1';
                bit_cntr<="001010";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='1';
                sdout<='1';
                wait for 100 ns;

--bit_cntr =11
                chan_on<='1';
                bit_cntr<="001011";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='1';
                sdout<='1';
                wait for 100 ns;
```

```
--bit_cntr =12
                    chan_on<='1';
                    bit_cntr<="001101";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                   wait for 100 ns;

--bit_cntr =13
                    chan_on<='1';
                   bit_cntr<="001101";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                   wait for 100 ns;

--bit_cntr =14
                    chan_on<='1';
                    bit_cntr<="001110";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                   wait for 100 ns;

--bit_cntr =15
                    chan_on<='1';
                   bit_cntr<="001111";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                    wait for 100 ns;


--bit_cntr =16
                    chan_on<='1';
                    bit_cntr<="010000";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='0';
                    sdout<='1';
                      wait for 100 ns;

--bit_cntr =17
                    chan_on<='1';
                    bit_cntr<="010001";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                   wait for 100 ns;

--bit_cntr =18
                    chan_on<='1';
                     bit_cntr<="010010";
                    subcycle_cntr<="10";
                    chan_sel<='1';
                    rd<='0';
                    sdout<='1';
                    wait;

                    END PROCESS;

-- *** End Test Bench - User Defined Section ***
END;
```

Description/ File name: TSTCHANNEL_RCVADC1.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

        COMPONENT channel
        PORT(
                clk : IN std_logic;
                reset : IN std_logic;
                chan_on : IN std_logic;
                bit_cntr : IN std_logic_vector(5 downto 0);
                subcycle_cntr : IN std_logic_vector(1 downto 0);
                chan_sel : IN std_logic;
                rd : IN std_logic;
                sdout : IN std_logic;
                adc_out : OUT std_logic_vector(19 downto 0);
                adc_out_rdy : OUT std_logic;
                adc_overrun : OUT std_logic
                );
        END COMPONENT;

        SIGNAL clk :  std_logic;
        SIGNAL reset :  std_logic;
        SIGNAL chan_on :  std_logic;
        SIGNAL bit_cntr :  std_logic_vector(5 downto 0);
        SIGNAL subcycle_cntr :  std_logic_vector(1 downto 0);
        SIGNAL chan_sel :  std_logic;
        SIGNAL rd :  std_logic;
        SIGNAL adc_out :  std_logic_vector(19 downto 0);
        SIGNAL adc_out_rdy :  std_logic;
        SIGNAL adc_overrun :  std_logic;
        SIGNAL sdout :  std_logic;
 SIGNAL Clock_cycle: natural := 0;

BEGIN

        uut: channel PORT MAP(
                clk => clk,
                reset => reset,
                chan_on => chan_on,
                bit_cntr => bit_cntr,
                subcycle_cntr => subcycle_cntr,
                chan_sel => chan_sel,
                rd => rd,
                adc_out => adc_out,
                adc_out_rdy => adc_out_rdy,
                adc_overrun => adc_overrun,
                sdout => sdout
        );

  CLOCK: process
  begin
    Clock_cycle <= Clock_cycle + 1;
    Clk <= '1';
    wait for 41.5 ns;
    Clk <= '0';
    wait for 41.5 ns;
  end process;

-- *** Test Bench - User Defined Section ***

tb : PROCESS  --: rcv_adc ( receive data from ADC)

BEGIN

--bit_cntr =0
```

```
                reset <= '1';
                chan_on<='1';
               bit_cntr<="000000";
                subcycle_cntr<="10";
                chan_sel<='1';
                rd<='1';
                sdout<='1';
               wait for 100 ns;

--bit_cntr = 1
                reset <= '0';
                chan_on<='1';
                  bit_cntr<="000001";
                subcycle_cntr<="10";
                chan_sel<='1';
                rd<='0';
                sdout<='1';
               wait for 100 ns;

--bit_cntr = 2
                reset <= '0';
                chan_on<='1';
                  bit_cntr<="000010";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='0';
                sdout<='1';
               wait for 100 ns;


--bit_cntr = 3
                chan_on<='1';
                bit_cntr<="000011";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='1';
                sdout<='1';
                wait for 100 ns;

--bit_cntr =4
                chan_on<='1';
                  bit_cntr<="000100";
                subcycle_cntr<="10";
                chan_sel<='1';
                rd<='0';
                sdout<='0';
               wait for 100 ns;

--bit_cntr =5
                chan_on<='1';
               bit_cntr<="000101";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='1';
                sdout<='1';
                wait for 100 ns;

--bit_cntr =6
                chan_on<='1';
               bit_cntr<="000110";
                subcycle_cntr<="01";
                chan_sel<='0';
                rd<='1';
                sdout<='1';
               wait for 100 ns;

--bit_cntr =7
                chan_on<='0';
                bit_cntr<="000111";
               subcycle_cntr<="10";
```

```
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                    wait for 100 ns;
--bit_cntr =8
                    chan_on<='1';
                     bit_cntr<="001000";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='0';
                   wait for 100 ns;

--bit_cntr =9
                    chan_on<='1';
                     bit_cntr<="001001";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                   wait for 100 ns;

--bit_cntr =10
                    chan_on<='1';
                     bit_cntr<="001010";
                    subcycle_cntr<="11";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                    wait for 100 ns;

--bit_cntr =11
                    chan_on<='1';
                    bit_cntr<="001011";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                    wait for 100 ns;

--bit_cntr =12
                    chan_on<='1';
                     bit_cntr<="001101";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='0';
                    wait for 100 ns;

--bit_cntr =13
                   chan_on<='1';
                   bit_cntr<="001101";
                   subcycle_cntr<="10";
                   chan_sel<='0';
                   rd<='1';
                   sdout<='1';
                   wait for 100 ns;

--bit_cntr =14
                   chan_on<='0';
                    bit_cntr<="001110";
                   subcycle_cntr<="10";
                   chan_sel<='0';
                   rd<='1';
                   sdout<='1';
                   wait for 100 ns;

--bit_cntr =15
                   chan_on<='1';
                    bit_cntr<="001111";
```

```
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                    wait for 100 ns;


--bit_cntr =16
                    chan_on<='1';
                    bit_cntr<="010000";
                    subcycle_cntr<="00";
                    chan_sel<='0';
                    rd<='0';
                    sdout<='1';
                    wait for 100 ns;

--bit_cntr =17
                    chan_on<='1';
                    bit_cntr<="010001";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                    wait for 100 ns;

--bit_cntr =18
                    chan_on<='1';
                    bit_cntr<="010010";
                    subcycle_cntr<="10";
                    chan_sel<='1';
                    rd<='0';
                    sdout<='1';
                    wait;

                    END PROCESS;

-- *** End Test Bench - User Defined Section ***
END;
```

Description/ File name: TSTCHANNEL_READOUT.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

        COMPONENT channel
        PORT(
                clk : IN std_logic;
                reset : IN std_logic;
                chan_on : IN std_logic;
                bit_cntr : IN std_logic_vector(5 downto 0);
                subcycle_cntr : IN std_logic_vector(1 downto 0);
                chan_sel : IN std_logic;
                rd : IN std_logic;
                sdout : IN std_logic;
                adc_out : OUT std_logic_vector(19 downto 0);
                adc_out_rdy : OUT std_logic;
                adc_overrun : OUT std_logic
                );
        END COMPONENT;

        SIGNAL clk :  std_logic;
        SIGNAL reset :  std_logic;
        SIGNAL chan_on :  std_logic;
```

```
        SIGNAL bit_cntr :  std_logic_vector(5 downto 0);
        SIGNAL subcycle_cntr :  std_logic_vector(1 downto 0);
        SIGNAL chan_sel :  std_logic;
        SIGNAL rd :  std_logic;
        SIGNAL adc_out :  std_logic_vector(19 downto 0);
        SIGNAL adc_out_rdy :  std_logic;
        SIGNAL adc_overrun :  std_logic;
        SIGNAL sdout :  std_logic;
  Signal Clock_cycle: natural := 0;
BEGIN

        uut: channel PORT MAP(
                clk => clk,
                reset => reset,
                chan_on => chan_on,
                bit_cntr => bit_cntr,
                subcycle_cntr => subcycle_cntr,
                chan_sel => chan_sel,
                rd => rd,
                adc_out => adc_out,
                adc_out_rdy => adc_out_rdy,
                adc_overrun => adc_overrun,
                sdout => sdout
        );

  CLOCK: process
  begin
    Clock_cycle <= Clock_cycle + 1;
    Clk <= '1';
    wait for 41.5 ns;
    Clk <= '0';
    wait for 41.5 ns;
  end process;

-- *** Test Bench - User Defined Section ***
  tb : PROCESS
  BEGIN

        -- adc is full and read process will only be permitted
        --when bit counter = 19, (shift register is full) and
        -- and has not been read

        --bit_cntr =18

                reset <= '1';
                chan_on<='1';
               bit_cntr<="010010";
                subcycle_cntr<="10";
                chan_sel<='1';
                rd<='0';
                sdout<='1';
                wait for 100 ns;

        --bit_cntr =19

               reset <= '0';
                chan_on<='1';
               bit_cntr<="010011";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='0';
                sdout<='1';
               wait for 100 ns;



                 chan_on<='1';
               bit_cntr<="010011";
                subcycle_cntr<="10";
```

```
                chan_sel<='0';
                rd<='0';
                sdout<='1';
    wait;
END PROCESS;

-- *** End Test Bench - User Defined Section ***
END;
```

Description/ File name: TSTCHANNEL_READOUT1.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

        COMPONENT channel
        PORT(
                clk : IN std_logic;
                reset : IN std_logic;
                chan_on : IN std_logic;
                bit_cntr : IN std_logic_vector(5 downto 0);
                subcycle_cntr : IN std_logic_vector(1 downto 0);
                chan_sel : IN std_logic;
                rd : IN std_logic;
                sdout : IN std_logic;
                adc_out : OUT std_logic_vector(19 downto 0);
                adc_out_rdy : OUT std_logic;
                adc_overrun : OUT std_logic
                );
        END COMPONENT;

        SIGNAL clk :  std_logic;
        SIGNAL reset :  std_logic;
        SIGNAL chan_on :  std_logic;
        SIGNAL bit_cntr :  std_logic_vector(5 downto 0);
        SIGNAL subcycle_cntr :  std_logic_vector(1 downto 0);
        SIGNAL chan_sel :  std_logic;
        SIGNAL rd :  std_logic;
        SIGNAL adc_out :  std_logic_vector(19 downto 0);
        SIGNAL adc_out_rdy :  std_logic;
        SIGNAL adc_overrun :  std_logic;
        SIGNAL sdout :  std_logic;
  Signal Clock_cycle: natural := 0;
BEGIN

        uut: channel PORT MAP(
                clk => clk,
                reset => reset,
                chan_on => chan_on,
                bit_cntr => bit_cntr,
                subcycle_cntr => subcycle_cntr,
                chan_sel => chan_sel,
                rd => rd,
                adc_out => adc_out,
                adc_out_rdy => adc_out_rdy,
                adc_overrun => adc_overrun,
                sdout => sdout
        );

  CLOCK: process
  begin
    Clock_cycle <= Clock_cycle + 1;
    Clk <= '1';
    wait for 41.5 ns;
```

```
    Clk <= '0';
    wait for 41.5 ns;
  end process;

-- *** Test Bench - User Defined Section ***
  tb : PROCESS
  BEGIN


        -- adc is full and read process will only be permitted
        --when bit counter = 19, (shift register is full) and
        -- and has not been read


   --bit_cntr =18

                    reset <= '1';
                    chan_on<='1';
                    bit_cntr<="010010";
                    subcycle_cntr<="10";
                    chan_sel<='1';
                    rd<='0';
                    sdout<='1';
                    wait for 100 ns;

        --bit_cntr =19

                    reset <= '0';
                    chan_on<='1';
                     bit_cntr<="010011";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                   wait for 100 ns;

-- ADC data is ready if register is full and has not been
-- read yet, where channel select and read,rd =0
                    chan_on<='1';
                    bit_cntr<="010011";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='0';
                    sdout<='0';
                     wait for 100 ns ;

                    chan_on<='1';
                    bit_cntr<="010011";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                    wait for 100 ns ;

                    chan_on<='0';
                    bit_cntr<="010011";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='1';
                    sdout<='1';
                   wait for 100 ns ;

                  chan_on<='1';
                   bit_cntr<="010011";
                   subcycle_cntr<="10";
                   chan_sel<='1';
                   rd<='1';
                   sdout<='1';
                   wait for 100 ns;
```

```
                chan_on<='1';
                bit_cntr<="010011";
                subcycle_cntr<="10";
                chan_sel<='0';
                rd<='0';
                sdout<='1';
                wait;

                END PROCESS;
```

- *** End Test Bench - User Defined Section ***

Description/ File name: TSTCHANNEL_OVERRUN.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

        COMPONENT channel
        PORT(
                clk : IN std_logic;
                reset : IN std_logic;
                chan_on : IN std_logic;
                bit_cntr : IN std_logic_vector(5 downto 0);
                subcycle_cntr : IN std_logic_vector(1 downto 0);
                chan_sel : IN std_logic;
                rd : IN std_logic;
                sdout : IN std_logic;
                adc_out : OUT std_logic_vector(19 downto 0);
                adc_out_rdy : OUT std_logic;
                adc_overrun : OUT std_logic
                );
        END COMPONENT;

        SIGNAL clk :  std_logic;
        SIGNAL reset :  std_logic;
        SIGNAL chan_on :  std_logic;
        SIGNAL bit_cntr :  std_logic_vector(5 downto 0);
        SIGNAL subcycle_cntr :  std_logic_vector(1 downto 0);
        SIGNAL chan_sel :  std_logic;
        SIGNAL rd :  std_logic;
        SIGNAL adc_out :  std_logic_vector(19 downto 0);
        SIGNAL adc_out_rdy :  std_logic;
        SIGNAL adc_overrun :  std_logic;
        SIGNAL sdout :  std_logic;
  Signal Clock_cycle: natural := 0;
BEGIN

        uut: channel PORT MAP(
                clk => clk,
                reset => reset,
                chan_on => chan_on,
                bit_cntr => bit_cntr,
                subcycle_cntr => subcycle_cntr,
                chan_sel => chan_sel,
                rd => rd,
                adc_out => adc_out,
                adc_out_rdy => adc_out_rdy,
                adc_overrun => adc_overrun,
                sdout => sdout
        );

  CLOCK: process
```

```
  begin
    Clock_cycle <= Clock_cycle + 1;
    Clk <= '1';
    wait for 41.5 ns;
    Clk <= '0';
    wait for 41.5 ns;
  end process;

-- *** Test Bench - User Defined Section ***
  tb : PROCESS
  BEGIN


--to test overrun (when adc is full and the old contents
--have not been read out)
--when bit counter=chan_on =1


                    reset <= '1';
                    chan_on<='1';
                   bit_cntr<="010011";  --adc is full when bit counter=19
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='0';
                    sdout<='1';
                    wait for 100 ns;

                    reset <= '0';
                    chan_on<='1';
                   bit_cntr<="010011";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='0';
                    sdout<='1';
                  wait for 100 ns ;

                  chan_on<='1';
                   bit_cntr<="000001";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                   rd<='0';
                    sdout<='1';
                    wait for 100 ns;


                  chan_on<='1';
                  bit_cntr<="000001";
                    subcycle_cntr<="10";
                    chan_sel<='0';
                    rd<='0';
                    sdout<='1';
                  wait;


                    END PROCESS;

-- *** End Test Bench - User Defined Section ***
END;
```

Description/ File name: TSTCHANNEL_OVERRUN1.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;
```

```
ARCHITECTURE behavior OF testbench IS

        COMPONENT channel
        PORT(
                clk : IN std_logic;
                reset : IN std_logic;
                chan_on : IN std_logic;
                bit_cntr : IN std_logic_vector(5 downto 0);
                subcycle_cntr : IN std_logic_vector(1 downto 0);
                chan_sel : IN std_logic;
                rd : IN std_logic;
                sdout : IN std_logic;
                adc_out : OUT std_logic_vector(19 downto 0);
                adc_out_rdy : OUT std_logic;
                adc_overrun : OUT std_logic
                );
        END COMPONENT;

        SIGNAL clk :  std_logic;
        SIGNAL reset :  std_logic;
        SIGNAL chan_on :  std_logic;
        SIGNAL bit_cntr :  std_logic_vector(5 downto 0);
        SIGNAL subcycle_cntr :  std_logic_vector(1 downto 0);
        SIGNAL chan_sel :  std_logic;
        SIGNAL rd :  std_logic;
        SIGNAL adc_out :  std_logic_vector(19 downto 0);
        SIGNAL adc_out_rdy :  std_logic;
        SIGNAL adc_overrun :  std_logic;
        SIGNAL sdout :  std_logic;
  Signal Clock_cycle: natural := 0;
BEGIN

        unt: channel PORT MAP(
                clk => clk,
                reset => reset,
                chan_on => chan_on,
                bit_cntr => bit_cntr,
                subcycle_cntr => subcycle_cntr,
                chan_sel => chan_sel,
                rd => rd,
                adc_out => adc_out,
                adc_out_rdy => adc_out_rdy,
                adc_overrun => adc_overrun,
                sdout => sdout
        );

  CLOCK: process
  begin
    Clock_cycle <= Clock_cycle + 1;
    Clk <= '1';
    wait for 41.5 ns;
    Clk <= '0';
    wait for 41.5 ns;
  end process;

-- *** Test Bench - User Defined Section ***
  tb : PROCESS
  BEGIN


--to test overrun (when adc is full and the old contents
--have not been read out)
--when bit counter=chan_on =1



                reset <= '1';
                 chan_on<='1';
                  bit_cntr<="010011";  --adc is full when bit counter=19
                 subcycle_cntr<="10";
```

```
chan_sel<='0';
rd<='0';
sdout<='1';
wait for 100 ns;

reset <= '0';
chan_on<='1';
bit_cntr<="010011";
subcycle_cntr<="10";
chan_sel<='0';
rd<='0';
sdout<='1';
wait for 100 ns ;

chan_on<='1';
bit_cntr<="010011";
subcycle_cntr<="10";
chan_sel<='0';
rd<='0';
sdout<='0';
wait for 100 ns ;


chan_on<='1';
bit_cntr<="000001";  --overrun happen when bit cntr =1
subcycle_cntr<="10";
chan_sel<='0';
rd<='0';
sdout<='1';
wait for 100 ns;


chan_on<='0';
bit_cntr<="000001";
subcycle_cntr<="10";
chan_sel<='0';
rd<='1';
sdout<='0';
wait for 100 ns;
chan_on<='1';
bit_cntr<="000001";
subcycle_cntr<="10";
chan_sel<='1';
rd<='0';
sdout<='1';
wait for 100 ns;


chan_on<='1';
bit_cntr<="000011";
subcycle_cntr<="10";
chan_sel<='1';
rd<='1';
sdout<='1';
wait for 100 ns;

chan_on<='1';
bit_cntr<="000111";
subcycle_cntr<="10";
chan_sel<='1';
rd<='1';
sdout<='1';
wait;


END PROCESS;

-- *** End Test Bench - User Defined Section ***
END;
```

Description/ File name: CODEC_INTFC.VHD

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_unsigned.ALL;

ENTITY codec_intfc IS
GENERIC
(
ADC_WIDTH: positive := 20;
CHANNEL_DURATION: positive := 128 -- must be 128
);
PORT
(
-- interface I/O signals
clk: IN std_logic; -- clock input
reset: IN std_logic; -- synchronous active-high reset
lrsel: IN std_logic; -- select L/R channel for read
rd: IN std_logic; -- read from the codec ADC
ladc_out: OUT std_logic_vector(ADC_WIDTH-1 DOWNTO 0); -- L ADC
radc_out: OUT std_logic_vector(ADC_WIDTH-1 DOWNTO 0); -- R ADC

ladc_out_rdy: OUT std_logic; -- left ADC output ready to read
radc_out_rdy: OUT std_logic; -- right ADC output ready to read
adc_overrun: OUT std_logic; -- ADC overwritten before read

-- codec chip I/O signals
sclk: OUT std_logic; --serial data clock to codec
lrck: OUT std_logic; -- left/right codec channel select
sdout: IN std_logic -- serial input from codec ADC
);
END codec_intfc;

ARCHITECTURE codec_intfc_arch OF codec_intfc IS
CONSTANT yes: STD_LOGIC := '1';
CONSTANT no: STD_LOGIC := '0';
CONSTANT ready: STD_LOGIC := '1';
CONSTANT overrun: STD_LOGIC := '1';
CONSTANT left: STD_LOGIC := '0';
CONSTANT right: STD_LOGIC := '1';

COMPONENT clkgen
GENERIC
(
CHANNEL_DURATION: positive := 128 -- must be 128
);
PORT
(
-- interface I/O signals
clk: IN std_logic; -- clock input
reset: IN std_logic; -- synchronous active-high reset
-- codec chip clock signals
sclk: OUT std_logic; --serial data clock to codec
lrck: OUT std_logic; -- left/right codec channel select
bit_cntr: OUT std_logic_vector(5 DOWNTO 0);
subcycle_cntr: OUT std_logic_vector(1 DOWNTO 0)
);
END COMPONENT;

COMPONENT channel

GENERIC
(

ADC_WIDTH: positive := 20
);
PORT
(
-- interface I/O signals
```

```
clk: IN std_logic; -- clock input
reset: IN std_logic; -- synchronous active-high reset
chan_on: IN std_logic;
bit_cntr: IN std_logic_vector(5 DOWNTO 0);
subcycle_cntr: IN std_logic_vector(1 DOWNTO 0);
chan_sel: IN std_logic; -- select L/R channel for read/write
rd: IN std_logic; -- read from the codec ADC
adc_out: OUT std_logic_vector(ADC_WIDTH-1 DOWNTO 0); -- ADC output
adc_out_rdy: OUT std_logic; -- ADC output is ready to be read
adc_overrun: OUT std_logic; -- ADC overwritten before being read
-- codec chip I/O signals
sdout: IN std_logic -- serial input from codec ADC
);
END COMPONENT;


SIGNAL lrck_int: std_logic; -- internal L/R codec channel select
SIGNAL sclk_int: std_logic; -- internal  codec data shift

SIGNAL bit_cntr: std_logic_vector(5 DOWNTO 0);
SIGNAL subcycle_cntr: std_logic_vector(1 DOWNTO 0);

SIGNAL ladc_overrun: std_logic;
SIGNAL radc_overrun: std_logic;

SIGNAL lchan_sel: std_logic;
SIGNAL rchan_sel: std_logic;
SIGNAL lchan_on: std_logic;
SIGNAL rchan_on: std_logic;

BEGIN

u0: clkgen
GENERIC MAP
(
CHANNEL_DURATION=>CHANNEL_DURATION
)
PORT MAP
(
clk=>clk,
reset=>reset,
sclk=>sclk_int,
lrck=>lrck_int,
bit_cntr=>bit_cntr,
subcycle_cntr=>subcycle_cntr
);


lrck <= not (lrck_int); -- invert for inverter in XStend V1.3
sclk <= not (sclk_int);
lchan_sel <= YES WHEN lrsel=LEFT ELSE NO;
lchan_on <= YES WHEN lrck_int=LEFT ELSE NO;
u_left: channel
GENERIC MAP
(

ADC_WIDTH=>ADC_WIDTH
)
PORT MAP
(
clk=>clk,
reset=>reset,
chan_on=>lchan_on,
bit_cntr=>bit_cntr,
subcycle_cntr=>subcycle_cntr,
chan_sel=>lchan_sel,
rd=>rd,
adc_out=>ladc_out,
adc_out_rdy=>ladc_out_rdy,
adc_overrun=>ladc_overrun,
```

```
sdout=>sdout
);

rchan_sel <= YES WHEN lrsel=RIGHT ELSE NO;
rchan_on <= YES WHEN lrck_int=RIGHT ELSE NO;

u_right: channel
GENERIC MAP
(

ADC_WIDTH=>ADC_WIDTH
)
PORT MAP
(
clk=>clk,
reset=>reset,
chan_on=>rchan_on,
bit_cntr=>bit_cntr,
subcycle_cntr=>subcycle_cntr,
chan_sel=>rchan_sel,
rd=>rd,
adc_out=>radc_out,
adc_out_rdy=>radc_out_rdy,
adc_overrun=>radc_overrun,
sdout=>sdout
);

adc_overrun <= YES WHEN ladc_overrun=YES OR radc_overrun=YES
ELSE NO;

END codec_intfc_arch;
```

Description/ File name: TSTCODEC_INTFC.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

        COMPONENT codec_intfc
        PORT(
                -- interface I/O signals
clk: IN std_logic; -- clock input
reset: IN std_logic; -- synchronous active-high reset
lrsel: IN std_logic; -- select L/R channel for read
rd: IN std_logic; -- read from the codec ADC

ladc_out: OUT std_logic_vector(19 DOWNTO 0); -- L ADC
radc_out: OUT std_logic_vector(19 DOWNTO 0); -- R ADC

ladc_out_rdy: OUT std_logic; -- left ADC output ready to read
radc_out_rdy: OUT std_logic; -- right ADC output ready to read
adc_overrun: OUT std_logic; -- ADC overwritten before read


-- codec chip I/O signals
sclk: OUT std_logic;
lrck: OUT std_logic; -- left/right codec channel select

sdout: IN std_logic -- serial input from codec ADC
                );
        END COMPONENT;

SIGNAL clk :  std_logic;
SIGNAL reset :  std_logic;
```

```
SIGNAL lrsel:  std_logic; -- select L/R channel for read
SIGNAL rd:  std_logic; -- read from the codec ADC
SIGNAL ladc_out: std_logic_vector(19 DOWNTO 0); -- L ADC
SIGNAL radc_out:  std_logic_vector(19 DOWNTO 0); -- R ADC

SIGNAL ladc_out_rdy:  std_logic; -- left ADC output ready to read
SIGNAL radc_out_rdy:  std_logic; -- right ADC output ready to read
SIGNAL adc_overrun:  std_logic; -- ADC overwritten before read
SIGNAL lrck:  std_logic; -- left/right codec channel select
SIGNAL sclk:  std_logic;
SIGNAL sdout: std_logic;
Signal Clock_cycle: natural := 0;

BEGIN

        uut: codec_intfc PORT MAP(
                clk => clk,
                reset => reset,
                lrsel=>     lrsel,
                rd=>rd,
ladc_out=>ladc_out,
radc_out => radc_out,
ladc_out_rdy =>ladc_out_rdy,
radc_out_rdy => radc_out_rdy,
adc_overrun=>  adc_overrun,
lrck => lrck,
sclk=>sclk,
sdout  => sdout
            );

  CLOCK: process
  begin
    Clock_cycle <= Clock_cycle + 1;
    clk <= '1';
    wait for 41.5 ns;
    clk <= '0';
    wait for 41.5 ns;
  end process;

-- *** Test Bench - User Defined Section ***
  tb : PROCESS
  BEGIN


                reset <= '1';
                rd<='0';                -- rd= output the value store in
                                        --SR connected to ADC----
                lrsel<='0';             --select left channel for
                                        --read operation----

                sdout<='1';             -- shift serial data from codec ADC--
                wait for 100 ns;


                reset <= '0';
                rd<='0';
                lrsel<='0';
                sdout<='1';
                wait ;
        END PROCESS;

-- *** End Test Bench - User Defined Section ***
END;
```

Description/ File name: CLOCK_DIVIDER.VHD

```
-- clock divider
```

```
-- divide 12MHz -> 1Hz
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY divider IS
        PORT (
        clock_12Mhz                     : IN    STD_LOGIC;
                clock_1MHz                      : OUT   STD_LOGIC;
        clock_100KHz    : OUT   STD_LOGIC;
                clock_10KHz                     : OUT   STD_LOGIC;
                clock_1KHz                      : OUT   STD_LOGIC;
                clock_100Hz                     : OUT   STD_LOGIC;
                clock_10Hz                      : OUT   STD_LOGIC;
                clock_1Hz               : OUT   STD_LOGIC);

END divider;

ARCHITECTURE comp OF divider IS

        SIGNAL count_1Mhz: STD_LOGIC_VECTOR(4 DOWNTO 0);
 SIGNAL count_100Khz, count_10Khz, count_1Khz : STD_LOGIC_VECTOR(2 DOWNTO 0);
        SIGNAL count_100hz, count_10hz, count_1hz : STD_LOGIC_VECTOR(2 DOWNTO 0);
        SIGNAL clock_1Mhz_int, clock_100Khz_int, clock_10Khz_int, clock_1Khz_int: STD_LOGIC;
        SIGNAL clock_100hz_int, clock_10Hz_int, onesec_int : STD_LOGIC;

BEGIN
        PROCESS
        BEGIN

-- Divide by 12

 WAIT UNTIL clock_12Mhz'EVENT and clock_12Mhz = '1';
 IF count_1Mhz < 11 THEN
                count_1Mhz <= count_1Mhz + 1;
 ELSE
   count_1Mhz <= "00000";
 END IF;
 IF count_1Mhz <7 THEN
   clock_1Mhz_int <= '0';
 ELSE
   clock_1Mhz_int <= '1';
 END IF;


clock_1Mhz <= clock_1Mhz_int;
clock_100Khz <= clock_100Khz_int;
clock_10Khz <= clock_10Khz_int;
clock_1Khz <= clock_1Khz_int;
clock_100hz <= clock_100hz_int;
clock_10hz <= clock_10hz_int;
clock_1Hz <= onesec_int;
        END PROCESS;

-- Divide by 10
        PROCESS
        variable startup : natural;
        BEGIN
        if startup = 0 then
        count_100Khz<="000";
        clock_100Khz_int<='0';
        startup:=1;
        end if;
        WAIT UNTIL clock_1Mhz_int'EVENT and clock_1Mhz_int = '1';
        IF count_100Khz /= 4 THEN
        count_100Khz <= count_100Khz + 1;
        ELSE
        count_100khz <= "000";
        clock_100Khz_int <= NOT clock_100Khz_int;
```

```
END IF;
END PROCESS;

-- Divide by 10
PROCESS
variable startup : natural;
BEGIN
if startup = 0 then
count_10Khz<="000";
clock_10Khz_int<='0';
startup:=1;
end if;
WAIT UNTIL clock_100Khz_int'EVENT and clock_100Khz_int = '1';
IF count_10Khz /= 4 THEN
count_10Khz <= count_10Khz + 1;
ELSE
count_10khz <= "000";
clock_10Khz_int <= NOT clock_10Khz_int;
END IF;
END PROCESS;

-- Divide by 10
PROCESS
variable startup : natural;
BEGIN
if startup = 0 then
count_1Khz<="000";
clock_1Khz_int<='0';
startup:=1;
end if;
WAIT UNTIL clock_10Khz_int'EVENT and clock_10Khz_int = '1';
IF count_1Khz /= 4 THEN
count_1Khz <= count_1Khz + 1;
ELSE
count_1khz <= "000";
clock_1Khz_int <= NOT clock_1Khz_int;
END IF;
END PROCESS;

-- Divide by 10
PROCESS
variable startup : natural;
BEGIN
if startup = 0 then
count_100hz<="000";
clock_100hz_int<='0';
startup:=1;
end if;
WAIT UNTIL clock_1Khz_int'EVENT and clock_1Khz_int = '1';
IF count_100hz /= 4 THEN
count_100hz <= count_100hz + 1;
ELSE
count_100hz <= "000";
clock_100hz_int <= NOT clock_100hz_int;
END IF;
END PROCESS;

-- Divide by 10
PROCESS
variable startup : natural;
BEGIN
if startup = 0 then
count_10hz<="000";
clock_10hz_int<='0';
startup:=1;
end if;
WAIT UNTIL clock_100hz_int'EVENT and clock_100hz_int = '1';
IF count_10hz /= 4 THEN
count_10hz <= count_10hz + 1;
ELSE
```

```
        count_10hz <= "000";
        clock_10hz_int <= NOT clock_10hz_int;
                END IF;
        END PROCESS;

        -- Divide by 10
        PROCESS
        variable startup : natural;
        BEGIN
        if startup = 0 then
        count_1hz<="000";
        onesec_int<='0';
        startup:=1;
        end if;
        WAIT UNTIL clock_10hz_int'EVENT and clock_10hz_int = '1';
        IF count_1hz /= 4 THEN
        count_1hz <= count_1hz + 1;
        ELSE
        count_1hz <= "000";
        onesec_int <= NOT onesec_int;
        END IF;
        END PROCESS;

END comp;
```

Description/ File name: TSTCLOCK_DIVIDER.VHD

```
LIBRARY ieee;

USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

        COMPONENT divider
        PORT (clock_12Mhz                       : IN STD_LOGIC;
                clock_1MHz                      : OUT   STD_LOGIC;
                clock_100KHz                    : OUT   STD_LOGIC;
                clock_10KHz                     : OUT   STD_LOGIC;
                clock_1KHz                      : OUT   STD_LOGIC;
                clock_100Hz                     : OUT   STD_LOGIC;
                clock_10Hz                      : OUT   STD_LOGIC;
                clock_1Hz                       : OUT   STD_LOGIC);
        END COMPONENT;


        SIGNAL clock_12Mhz                      :       STD_LOGIC;
        SIGNAL  clock_1MHz                      :       STD_LOGIC;
        SIGNAL  clock_100KHz                    :       STD_LOGIC;
        SIGNAL  clock_10KHz                     :       STD_LOGIC;
        SIGNAL  clock_1KHz                      :       STD_LOGIC;
        SIGNAL  clock_100Hz                     :       STD_LOGIC;
        SIGNAL  clock_10Hz                      :       STD_LOGIC;
        SIGNAL  clock_1Hz                       :       STD_LOGIC;
        Signal Clock_cycle: natural := 0;

BEGIN

        div: divider PORT MAP(
                        clock_12mhz => clock_12mhz,
                        clock_1mhz => clock_1mhz,
                        clock_100khz=> clock_100khz,
                        clock_10khz => clock_10khz,
                        clock_1khz => clock_1khz,
                        clock_100hz => clock_100hz,
                        clock_10hz => clock_10hz,
```

```
                clock_1Hz => clock_1Hz);

    CLOCK: process
    begin
      Clock_cycle <= Clock_cycle + 1;
      clock_12mhz <= '1';
      wait for 41.5 ns;
      clock_12mhz <= '0';
      wait for 41.5 ns;
    end process;


-- *** Test Bench - User Defined Section ***


-- *** End Test Bench - User Defined Section ***
END;
```

Description/ File name: LED.VHD

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity leddcd_behavioral is
    Port ( d : in std_logic_vector(3 downto 0);
           s : out std_logic_vector(6 downto 0));
end leddcd_behavioral;

architecture Behavioral of leddcd_behavioral is

begin
s           <="1110111" when d="0000" else
                "0010010" when d="0001" else
                "1011101" when d="0010" else
                "1011011" when d="0011" else
                "0111010" when d="0100" else
                "1101011" when d="0101" else
                "1101111" when d="0110" else
                "1010010" when d="0111" else
                "1111111" when d="1000" else
                "1111011" when d="1001" else
                "1111110" when d="1010" else
                "0101111" when d="1011" else
                "0001101" when d="1100" else
                "0011111" when d="1101" else
                "1101101" when d="1110" else
                "1101100";
end Behavioral;
```

Description/ File name: TSTLED.VHD

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavior OF testbench IS

        COMPONENT leddcd_behavioral
        PORT(
                d : IN std_logic_vector(3 downto 0);
                s : OUT std_logic_vector(6 downto 0)
                );
        END COMPONENT;
```

```
        SIGNAL d :  std_logic_vector(3 downto 0);
        SIGNAL s :  std_logic_vector(6 downto 0);
BEGIN

        uut: leddcd_behavioral PORT MAP(
                d => d,
                s => s
        );

-- *** Test Bench - User Defined Section ***
  tb : PROCESS
  BEGIN
  d <= "0000";
      wait for 50ns; -- will wait forever
  d <= "0001";
      wait for 50ns; -- will wait forever
  d <= "0010";
      wait for 50ns; -- will wait forever


                        END PROCESS;
-- *** End Test Bench - User Defined Section ***

END;
```

References

1) [Peter, 1990] " The VHDL Cookbook", University of Adelaide, South Australia, Peter J. Ashden, 1990.
2) [Douglas, 1998] "VHDL (Computer Hardware Description Language) ", Douglas L. Perry, Mc Graw-Hill Co Singapore, 1998.
3) [Morris, 2000] "Logic and Computer Design Fundamentals", M. Morris Mano and Charles R. Kime, Upper Saddle River, New Jersey, Prentice Hall, Inc, 2000
4) [Andrew,1987] "Robotics and Artificial Intelligence: An Introduction to Applied Machine Intelligence", Andrew C. Staugaard Jr, Prentice Hall, Inc, 1987
5) [Bahl, 1986] " "Maximum Mutual Information Estimation of Hidden Markov Model Parameters for Speech Recognition", Bahl. L., Brown, P., 1986.
6) [Hirschberg, 1993] "Pitch accent in context: Predicting intonational prominence from text - Artificial Intelligence" , Hirschberg. J, 1993.
7) [Ian, 1982] "Principles of Computer Speech, London", Ian H. Witten, Academic Press Inc LTD, 1982.
8) "Trends in Speech Recognition", Wayne A. Lea, Englewood Cliffs, New Jersey, Prentice-Hall, 1980.
9) "Visible Speech", Ralph K. Poter, George A Kopp, Hakriet Greens Kopp, New York, Dover Publications, 1966
10) "Language Independent and Language Adaptive Acoustic Modeling. In *Speech Communication*", T. Schultz and A. Waibel. Vol 35, August 2001.
11) "Pitch accent in context: Predicting intonational prominence from text-Artificial Intelligence", Hirschberg J, 1993.
12) http://whatis.techtarget.com/definition/0,,sid9_gci213760,00.html
13) www.xess.com
14) http://wombat.doc.ic.ac.uk/foldoc
15) http://www.gear21.com/speech/html/inside.html
16) www.sigda.org/Archives/ProceedingArchives/Compendiums/Compendium2001/papers/2001/aspdac01/pdffiles/5c_2.pdf

17) www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1999f/UART/uart.h
18) www.cirrus.com
19) www.xilinx.com/ise
20) www.VoiceActivation.com
21) www.st.com/voice
22) http://www.DigitalCoreDesign.com
23) http://www.dcd.pl
24) www.sensoryinc.com
25) www.es.isy.liu.se/courses/TSTE90/ download/HW-info_TSTE90.pdf
26) www.courses.ece.uiuc.edu/ece311/lectures/lecture17.ppt
27) www.egr.msu.edu/annweb/papers/blind_separation/ ieee_cas99_bsr.pdf
28) www.utdallas.edu/~loizou/loires.html
29) www.bioid.com/sdk/docs/About_Preprocessing.htm
30) www.ece.uvic.ca/499/2002a/group05/product/specs.html